Proceedings for

International Workshop on

Tools for Managing Globally Distributed Software Development (TOMAG 2007)

+

Tool Support and Requirements Management in Distributed Project (REMIDI 2007)

Munich, Germany
August 27, 2007

Edited by Jos Van Hillegersberg, Frank Harmsen, Chintan Amrit, Dr. Eva Geisberger, Patrick Keil, Marco Kuhrmann

Title: TOMAG+REMIDI 2007 Proceedings. The Seventh International Conference of Computer Ethics: Philosophical Enquiry
Author: Jos Van Hillegersberg, Frank Harmsen, Chintan Amrit, Dr. Eva Geisberger, Patrick Keil, Marco Kuhrmann (eds.)
ISSN: 1574-0846
Publisher: Center for Telematics and Information Technology (CTIT), Enschede, the Netherlands

As part of

International Conference on
Global Software Engineering, **ICGSE 2007**

Program Committee (TOMAG)

Jos van Hillegersberg,	University of Twente
Frank Harmsen,	Cap Gemini
Kuldeep Kumar,	Florida International University
Mehmet Aksit,	University of Twente
Richard Welke,	Georgia State University
Matti Rossi,	Helsinki School of Economics and Business Administration
Gert-Jan de Vreede,	University of Nebraska at Omaha
M. E. Iacob,	University of Twente
Robert Slagter,	Telematica Institute
Harry Julsing,	Mithun Training & Consulting BV
Tobias Kuipers,	Software Improvement Group
Joost Visser,	Software Improvement Group

Program Committee (REMIDI)

Matthew Bass,	Carnegie Mellon University
Stefan Biffl,	TU Wien
Manfred Broy,	TU München
Mathai Joseph,	Tata Consultancy Services
Thomas Klingenberg,	microTOOL GmbH
Vesna Mikulovic,	Siemens AG Austria
Jürgen Münch,	Fraunhofer IESE
Ita Richardson,	University of Limerick
Bernhard Schätz,	TU München
Gernot Stenz,	TU München

Table of Contents

1. Introduction TOMAG 2007……..1

2. Monitoring the Quality of Outsourced Software..……3
Kuipers, T; Visser, J and de Vries, G

3. MAIS: an awareness mechanism for change identification on shared models….....12
de *M. Lopes, M A.; Werner, C. M. L. and Mangan, M. A. S*

4. Exploring Coordination structures in Open source Software Development.....……22
Amrit, C; Hegeman, J.H. and van Hillegersberg, J

5. A framework for designing integrated tool support for globally distributed software…….30
development teams
Herrera, M and van Hillegersberg, J

6. Introduction REMIDI 2007……37

7. A Sensitivity Analysis Approach to Select IT-Tools for Global Development Projects…….40
Laurent, C

8. Requirements Management Infrastructures in Global Software Development……46
-Towards Application Lifecycle Management with Role-Oriented In-Time Notification
Heindl, M.; Reinisch, F. and Biffl S.

9. Communication Tools in Globally Distributed Software Development Projects……..53
Niinimäki, T

10 A Groupware System for Distributed Collaborative Programming: Usability Issues……..59
and Lessons Learned
Bravo, C; Duque, R; Gallardo, J; García, J and García, P

1st International Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007)

Chintan Amrit
University of Twente
c.amrit@utwente.nl

Jos van Hillegersberg
University of Twente
j.vanHillegersberg@utwente.nl

Frank Harmsen
Cap Gemini
frank.harmsen@capgemini.com

Abstract

The advent of global distribution of software development has made managing collaboration and coordination among developers more difficult due to various reasons including physical distance, differences in time, cultural differences etc. A nearly total absence of informal communication among developers makes coordinating work in a globally distributed setting more critical. The goal of this workshop is to provide an opportunity for researchers and industry practitioners to explore both the state-of-the art in tools and methodologies for managing global software development (GSD).

1. Introduction

Large scale software development is an inherently collaborative, team based process, and hence requires coordination and control in order to make it successful. The advent of global distribution of software development has made managing this collaboration more difficult due to various reasons including physical distance, differences in time, cultural differences etc. Although research on global software development argues the use of communication technologies to alleviate problems caused by separation of workers in time and space, studies have often found them to be not as effective as publicized. A nearly total absence of informal communication among developers makes coordinating work in a globally distributed setting more critical.
The goal of this workshop is to provide an opportunity for researchers and industry practitioners to explore both the state-of-the art in tools and methodologies for managing global software development (GSD). The workshop will foster interaction between practitioners and researchers and help grow a community of interest in this area. Practitioners experiencing challenges in GSD are invited to share their concerns and successful solutions. Practitioners will have the opportunity to gain a better understanding of the key issues facing other practitioners and share their work in progress with others in the field.
In this workshop we examine the technologies that go beyond mere communication technologies and which aim to manage the coordination problems encountered in globally distributed development. We are particularly interested in empirical research (case studies) on globally distributed projects such as open source, commercial, and government projects. Such projects are often dominated by social, rather than just technical, issues, and so would significantly benefit from appropriate tool support.

2. Workshop Scope and Theme

The main theme of this workshop is Tools for Managing Globally distributed software development. . We are particularly interested in empirical research (case studies) on globally distributed projects such as open source, commercial, and government projects. Such projects are often dominated by social, rather than just technical, issues, and so would significantly benefit from appropriate tool support.

Some of the topics of interest to this workshop are:

- Communication, collaboration, and awareness tools for globally distributed software development
- Visualization systems to support social aspects of globally distributed software development

- Evaluation techniques for studying the effectiveness and impact of collaborative software development tools
- CASE and Requirement tools for managing globally distributed systems
- Project management tools and environments for globally distributed teams
- Communities of interest, communities of practice, knowledge sharing and organizational learning
- Interaction in large scale online communities supporting collaboration in local and distributed communities

3. Workshop Presentations

Apart from the Key Note speech of Alan Hartman, of IBM Haifa, Israel, the workshop has 3 full papers and one short paper presentation.

The workshop accepted both full papers of maximum 8 pages and short pages of 4 pages maximum.
We received 3 full papers and two short papers. Among the full papers Lopes M., Werner C. and Mangan Marco describe an awareness mechanism prototype for collaborative modeling. Through a preliminary case study of 10 university students they try and validate the tool design and usage.

Amrit C. and Hillegersberg J. describe a methodology to track globally distributed development. In a case study of 3 open source projects they use a clustering mechanism to see how the tasks of the developers change over a period of time.

Kuipers T., Gejon V. and Visser J. present a tool-based method for monitoring software in outsourcing situations. Through the 3 case studies, the authors have shown the usefulness of the method to maintain the software quality through continuous monitoring.

Among the short papers, Hillegersberg J. and Herrera M. describe a framework for designing integrated tool support for globally distributed software development teams. Hegeman J.H. and Amrit C. describe a tool which represents the developers and the software they are developing through clustering.

4. Summary and Discussion

One of the goals of this workshop was to have a discussion about the current practices as well as the future of Tool development in a globally distributed software development scenario. In order to have a broader platform for discussion we have a joint session with REMIDI 2007.

Monitoring the Quality of Outsourced Software

Tobias Kuipers
Software Improvement Group
The Netherlands
Email: t.kuipers@sig.nl

Joost Visser
Software Improvement Group
The Netherlands
Email: j.visser@sig.nl

Gerjon de Vries
Software Improvement Group
The Netherlands
Email: g.devries@sig.nl

***Abstract*—Outsourcing application development or maintenance, especially offshore, creates a greater need for hard facts to manage by. We have developed a tool-based method for software monitoring which has been deployed over the past few years in a diverse set of outsourcing situations. In this paper we outline the method and underlying tools, and through several case reports we recount our experience with their application.**

I. Introduction

Outsourcing of application development or maintenance brings about an interesting dilemma. One the one hand, outsourcing promisses cost reduction and increased focus on core business. The vendor organization specializes in software engineering capabilities and realizes scaling benefits. On the other hand, the loss of technical expertise at the client organization leads to loss of control over the quality of the delivered product which, in turn, leads to loss of efficiency and increased costs. To prevent that costs, deadlines, and functionality slip out of control, the remote management of outsourced projects must be grounded in factual technical knowledge of the outsourced system. Is there a way out of this dilemma?

In this paper, we argue that this outsourcing dilemma can be resolved by performing tool-assisted monitoring of the quality of outsourced software. Such software monitoring is a highly specialized activity that supports IT management by translating technical findings to actionable recommendations. To avoid the need for in-house technical know-how, this highly specialized activity of monitoring outsourced software can in turn be outsourced to a third, independent party.

We have developed a tool-based method for software monitoring which has been deployed over the past few years in a diverse set of outsourcing situations. In this paper we outline the method and underlying tools, and through several case reports we recount our experience with their application.

The paper is structured as follows. Section II provides a global overview of the tool-based method for software monitoring that we have developed previously [1]. Section III high-lights the tools that support the method, while Section IV focusses on the quality model it employs. In Section V, we share our experiences with applying the method in the form of three case reports. These reports cover various application scenarios and various software technology platforms. The paper is concluded in Section VI, where we summarize our contributions and reflect on lessons learned and on the value of software monitoring on a more generalized level.

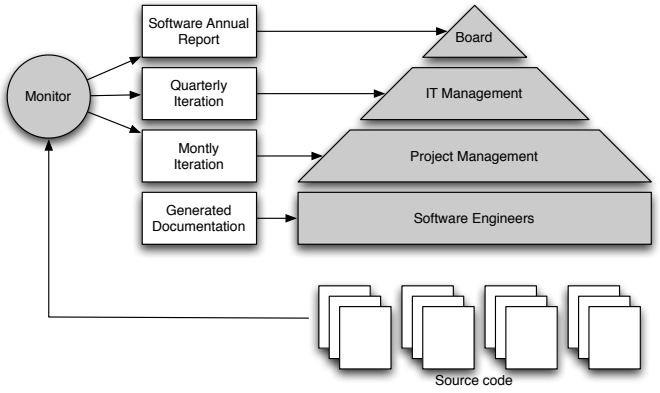


Fig. 1. The various deliverables of software monitoring and their relationships to management levels. On the basis of source code analysis, insight in the technical quality of software systems or entire portfolios is provided at regular intervals to project managers, IT management, and general management.

II. Software Monitoring

Previously, we have developed and described a tool-based method for software monitoring which consists of a cycle of activities designed to drive continuous improvement in software products and processes [1]. An overview of the method is provided in Figure 1.

A. Source code analysis

The basis of monitoring is the frequent analysis of all source code in an automated manner. We have developed a suite of tools, dubbed the Software Analysis Toolkit (SAT), which contains components for parsing, flow analysis, and metric extraction for a wide range of programming languages. The SAT has been designed to be highly scalable and highly customizable. It is suitable for processing software portfolios of many millions of lines of code. We continuously extend the SAT with support for further languages.

A dynamic web portal with all extracted software metrics is available to all stake holders in the system or portfolio.

B. Scope

The scope of software monitoring is flexible, both in duration and in the number of systems being monitored. In some cases, only a single system is monitored, but more commonly all systems with a particular technology footprint (e.g. mainframe systems, or .Net systems) are under scrutiny. When the scope extends to all systems, we use the term software portfolio monitoring. In some cases, monitoring is

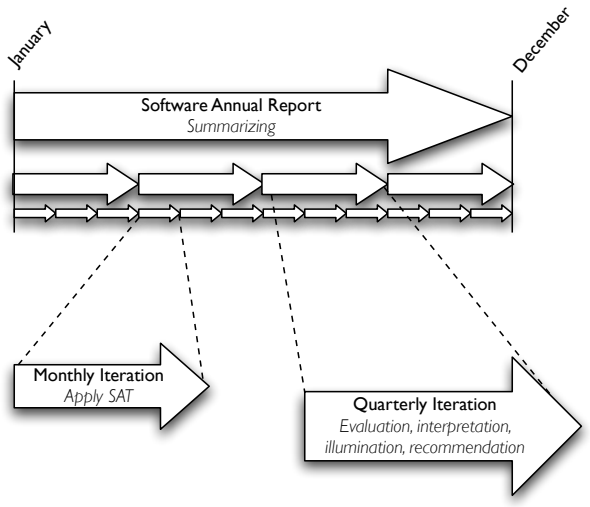


Fig. 2. The various nested iterations of software monitoring.

initiated at the start of system development, and ended at first delivery. In other cases, systems are monitored in their maintenance or renovation phase.

C. Deliverables

The information extracted from source code is combined with information obtained from secondary sources, such as documentation and interviews with stake holders. On the basis of this combination, insight into the software is provided to various levels of management at different frequencies. With high frequency (typically monthly), fairly detailed information about individual systems is communicated to project managers. With medium frequency, more aggregated information is presented and discussed at the level of overall IT management. With low frequency, the monitoring information of an entire year is compressed into an annual software report, to be presented at board level.

Note that the various deliverables are not limited to simply communicating measurement values. The results of source code analysis are interpreted by experts, followed by evaluation and recommendations. Where feasible, targets are set for quality improvement. Thus, a significant consultancy effort is mounted to incorporate the lessons to be learned from software measurements into the software management processes.

Below we discuss the various iterations and their deliverables in more detail.

D. Iterations

The three nested iterations of the monitoring methodology are illustrated in more detail in Fig. 2. Though the typical duration of the shortest iteration is one month, shorter and longer time spans are also used. The yearly iteration is optional, and is typically used only when a significant part of the software portfolio is being monitored.

1) Monthly iteration: In the inner iteration, the Software Analysis Toolkit is applied to the selected software systems or entire portfolio, resulting in a large number of basic facts about the code. These facts include metrics, dependency information, and detected violations of programming conventions or standards. All these facts are collected into a data repository. From this repository, reports are generated that present the facts in a human digestible fashion. This means that the data is appropriately grouped and filtered, and visualized in graphs and charts that meet the information needs of assessment experts, project managers, and other stake holders.

2) Quarterly iteration: Every three months, the technical data gathered in the inner iterations is interpreted and evaluated by assessment experts. Also, the data is related to other information elicited in workshops and interviews. The findings are presented to IT management together with recommendations about how to react to the findings.

By interpretation, we mean that various selections of the data are combined and contrasted to discover for instance trends, correlations, and outliers. For example, based on the fact that certain modules have exceeded a certain complexity threshold, an assessment expert might hypothesize that these modules implement several related functions in a tangled fashion. He might discover that the database dependency information for these modules corroborates his hypothesis. Finally, he may take a small sample from these modules, inspect their code and verify that his hypothesis is indeed true.

By evaluation, we mean that the expert makes value judgments about the software system or portfolio. The judgments are based on best practices reported in the literature, on published quality standards, comparisons with industry best and average, and so on. In Section IV, we provide further insight into the structured method we use for software quality evaluation according to the ISO/IEC 9126 software product quality model [2].

The evaluation and interpretation of technical data, as well as elicitation of IT-related business goals are instrumental in the most important element of the quarterly iteration: the drafting of recommendations. These recommendations are of various kinds. They can be detailed, short-term recommendation, such as redesigning a particular interface, migrating particular persistent data from hierarchical to relational storage, or upgrading a particular third-party component. On the other hand, some recommendations may have a more general, long term character, such as integrating two functionally similar, but technically distinct systems, or reducing the procedural character of the object-oriented portions of the portfolio.

The deliverable of the quarterly iteration is a presentation of findings, evaluation, and recommendations to IT management in a workshop dedicated to that purpose.

3) Annual iteration: Every year, the deliverables of the monthly and quarterly iterations are summarized in an Annual Software Report. The intended audience of this report is the general management of the company, which is not necessarily IT-savvy. For this reason, the software engineering experts that compile the report need to be able to explain IT issues in layman's terms. In addition to the summaries of the monthly and quarterly iterations, the Annual Software Report may include IT-related financial information, if available in sufficient detail.

III. Tool basis

In this section we provide a brief discussion of the tool support for source code analysis on which the monitoring approach is based.

The tools offer three overall pieces of functionality: gathering source code, performing static analysis on the code, and visualizing the analysis results. The components that implement analysis and visualization are frameworks into which various subcomponents can be inserted that implement individual analysis and visualization algorithms. A repository that persistently stores all information extracted from the sources is shared by the components for gathering, analysis, and visualization.

A. Source Manager

Source code files can be brought into the system in different ways. In some cases, a connection is made to the versioning system of the client, such that the upload procedure is fully automatic. In other cases, the technical situation or client preferences do now allow full automation. For these cases, a secure upload facility is provided which can be operated by the client via a standard web browser.

B. Analysis Components

Once source code files have been uploaded to the system, they will be analyzed statically by the analysis framework. Which analyses are available for the various source files depends on the specific client configuration.

Analysis components vary in their degree of sophistication and generality. Some components are applicable only to certain types of files. For instance, a component of control-flow reconstruction may implement an algorithm that works only for ANSI-Cobol-85. Other components are applicable more generally. For instance, a component for counting lines of code and comment could work for any language that employs one of the common comment conventions.

The amount of source code in a typical software portfolio ranges between 1 million and 100 million lines of code. Processing this code to obtain the basic monitoring data should under no circumstance take more than a few hours. The computational complexity of the implemented algorithms should be kept within bounds. In this sense, the analysis components must be scalable.

C. Visualization components

Basically, two categories of visualizations are available: charts and graphs. Both are highly parameterizable. We are not only interested in presenting data about software at a particular moment. We need to visualize the *evolution* of the software throughout time. Of course charts can be used for this purpose, where one of the axes represents time. Another instrument is the use of animations.

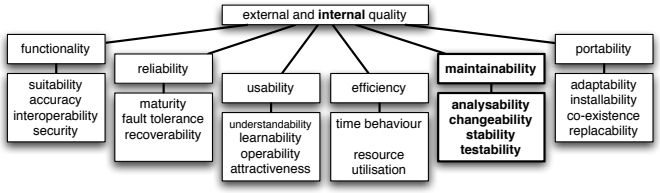


Fig. 3. Breakdown of the notions of internal and external software product quality into 6 main characteristics and 27 sub-characteristics (the 6 so-called compliance sub-characteristics are not shown). In this paper, we focus on the maintainability characteristic and its 4 sub-characteristics of analyzability, changeability, stability, and testability.

IV. A practical model of technical quality

The ISO/IEC 9126 standard [2] describes a model for software product quality that dissects the overall notion of quality into 6 main characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics are further subdivided into 27 sub-characteristics. This breakdown is depicted in Fig. 3. Furthermore, the standard provides a consensual inventory of metrics that can be used as indicators of these characteristics [3], [4]. The defined metrics provide guidance for *a posteriori* evaluation based on effort and time spent on activities related to the software product, such as impact analysis, fault correction, or testing. Remarkably, ISO/IEC 9126 does not provide a consensual set of measures for estimating maintainability on the basis of a system's *source code*.

Over the course of several years of management consultancy grounded in source code analysis, we have started to formulate a software quality model in which a set of well-chosen source-code measures are mapped onto the sub-characteristics of maintainability according to ISO/IEC 9126, following pragmatic mapping and ranking guidelines [5]. We briefly present this model.

A. Mapping source code properties quality aspects

The maintainability model we have developed links system-level maintainability characteristics to code-level measures in two steps. Firstly, it maps these system-level characteristics to properties on the level of source code, e.g. the *changeability* characteristic of a system is linked to properties such as *complexity* of the source code. Secondly, for each property one or more source code measures are determined, e.g. source code complexity is measured in terms of *cyclomatic complexity*.

Our selection of source code properties, and the mapping of system characteristics onto these properties is shown in Fig. 4. The notion of source code *unit* plays an important role in various of these properties. By a *unit*, we mean the smallest piece of code that can be executed and tested individually. In Java or C# a unit is a method, in C a unit is a procedure. For a language such as COBOL, there is no smaller unit than a program. Further decompositions such as sections or paragraphs are effectively labels, but are not pieces of code that are sufficiently encapsulated to be executed or tested individually.

ISO/IEC 9126 maintainability	source code properties: volume	complexity per unit	duplication	unit size	unit testing
analysability	x		x	x	x
changeability		x	x		
stability					x
testability		x		x	x

Fig. 4. Mapping system characteristics onto source code properties. The rows in this matrix represent the 4 maintainability characteristics according to ISO/IEC 9126. The columns represent code-level properties, such as *volume*, *complexity*, and *duplication*. When a particular property is deemed to have a strong influence on a particular characteristic, a cross is drawn in the corresponding cell.

The influence of the various source code properties on maintainability characteristics of software is as follows:

- Volume: The overall volume of the source code influences the analysability of the system.
- Complexity per unit: The complexity of the code units influences the system's changeability and its testability.
- Duplication: The degree of source code duplication influences analysability and changeability.
- Unit size: The size of units influences their analysability and testability and therefore of the system as a whole.
- Unit testing: The degree of unit testing influences the analysability, stability, and testability of the system.

This list of properties is not intended to be complete, or provide a watertight covering of the various system-level characteristics. Rather, they are intended to provide a minimal, non-controversial estimation of the main causative relationships between code properties and system characteristics. Intentionally, we only high-light the most influential causative links between source code properties and system characteristics. For instance, the absence of a link between volume and testability does not mean the latter is not influenced at all by the former, but rather that the influence is relatively minor.

B. Ranking

For ranking, we use the following simple scale for each property and characteristic: ++ / + / o / - / --. For various code-level properties we have defined straightforward guidelines for measuring and ranking them.

As an example, consider the property of *complexity*. The complexity property of source code refers to the degree of internal intricacy of the source code units from which it is composed. Since the unit is the smallest piece of a system that can be executed and tested individually, it makes sense to calculate the cyclomatic complexity on each unit. To arrive at a meaningful aggregation of the complexity values of the various unit of a system, we take the following categorization

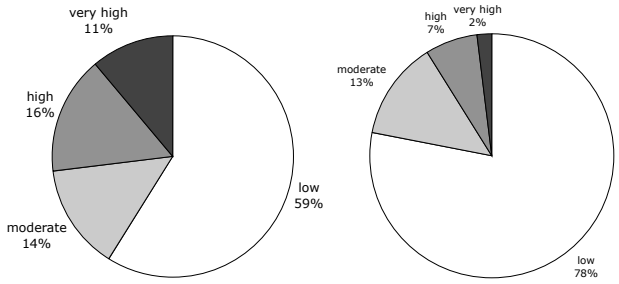


Fig. 5. Distribution of lines of code over the four complexity risk levels for two different systems. Regarding complexity, the leftmost system scores -- and the rightmost system scores -.

of units by complexity, provided by the Software Engineering Institute, into account [6]:

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
> 50	untestable, very high risk

Thus, from the cyclomatic complexity of each unit, we can determine its risk level. We now perform aggregation of complexities per unit by counting for each risk level what percentage of lines of code falls within units categorized at that level. For example, if, in a 10.000 LOC system, the high risk units together amount to 500 LOC, then the aggregate number we compute for that risk category is 5%. Thus, we compute relative volumes of each system to summarize the distribution of lines of code over the various risk levels. These complexity 'footprints' are illustrated in Fig. 5 for two different systems.

Given the complexity footprint of a system, we determine its complexity rating using the following schema:

	maximum relative LOC		
rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
--	-	-	-

Thus, to be rated as ++, a system can have no more than 25% of code with moderate risk, no code at all with high or very high risk. To be rated as +, the system can have no more than 30% of code with with moderate risk, no more than 5% with high risk, and no code with very high risk. A system that has more than 50% code with moderate risk or more than 15% with high or more than 5% with very high risk is rated as --.

For example, the system with the leftmost complexity profile of Fig. 5 will be rated as --, since it breaks both the 15% boundary for high risk code and the 5% boundary for very high risk code. The rightmost profile leads to a - rating, because it breaks the 0%, but not the 5% boundary for very high risk code.

Similar rating guidelines have been defined for other source code properties. Details can be found elsewhere [5]. The

boundaries and thresholds we defined are based on experience. During the course of evaluating numerous systems, these boundaries turned out to partition systems into categories that corresponded to expert opinions.

C. Practicality of the quality model

Our quality model exhibits a number of desirable properties.

- The measures are mostly technology independent. As a result, they can be applied to systems that harbour various kinds of languages and architectures.
- Each measure has a straightforward definition that is easy to implement and compute. Consequently, little up-front investment is needed to perform the measurement.
- Each measure is simple to understand and explain, also to non-technical staff and management. This facilitates communication to various stake holders in the system.
- The measures enable root-cause analysis. By giving clear clues regarding causative relations between code-level properties and system-level quality, they provide a basis for action.

Due to these properties, the model has proven to be practically usable in the context of software monitoring.

V. Case studies

Over the past few years, we have applied software monitoring in a wide range of management consultancy projects. In this section, we share some of our experiences in three anonymized case reports.

A. Curbing system erosion during maintenance

An organisation has automated a part of its primary business process in a software system some 10 to 15 years ago. A party that currently plays no role in the maintenance of the system built it. Over time, maintenance has passed through a number of organisations. The system is currently being operated and managed in a location in central Europe, and being maintained in South East Asia. The system owner (business requirements developer) is in a different location in western Europe. The system owner periodically requests specific features to be added to the system, and from time to time the system needs to be adapted to a changing hardware environment.

We were asked to monitor the maintenance of the system in order to improve management's control over the technical quality of the software and the associated costs of the maintenance process.

As a result of the monitoring activity, we had accurate insight into various system parameters, among which its *volume*. In Fig. 6, the volume of the system, measured in lines of code, is plotted for the 4 latest releases of the system, separated into C code, stored procedures (PL/SQL), and scripts. Note that an increase in the volume of C code of about 35% occurred between release $r1$ and release $r2$. Such increases are normal in development situations, but in the current maintenance situation, where the system had been more or less stable for a number of years, this amount of growth is remarkable. When we asked the various parties involved with the system what

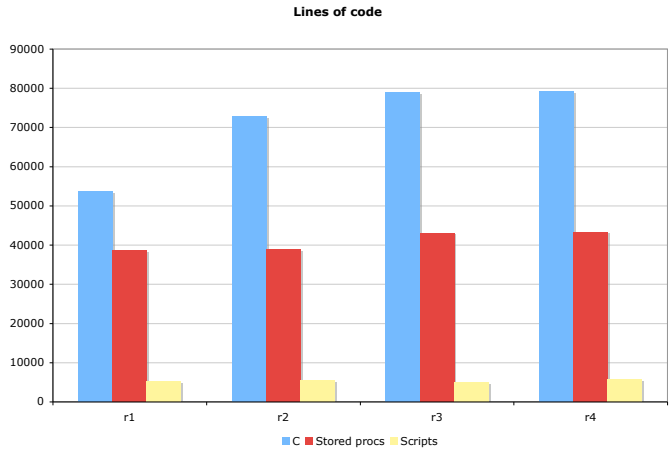


Fig. 6. System volume in lines of code for the latest 4 releases (Case V-A).

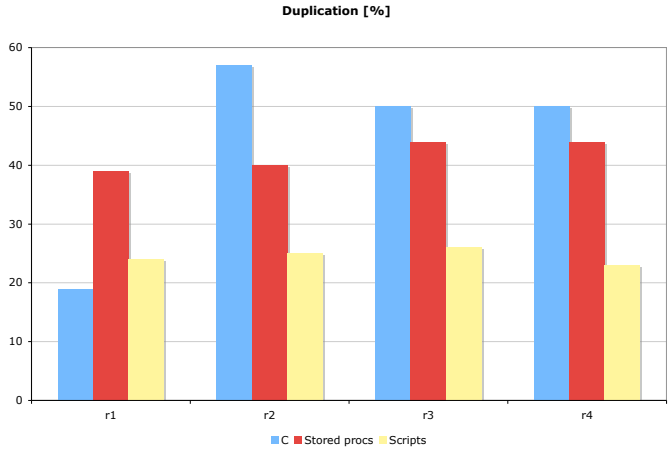


Fig. 7. Code duplication for the latest 4 releases (Case V-A).

could explain this growth, we were met with blank stares; nobody could come up with a reason why the system was growing so much.

Another parameter that we monitored is the amount of code duplication in the system. This is expressed as a percentage of code lines. A code line counts as duplicated when it participates in a block of code of at least 6 lines that occurs more than once. Apart from some compensation for spacing issues, we count exact duplicates. For the same 4 releases, the measurement values are plotted in Fig. 7. As it turned out, the unexplained growth of the C code between release $r1$ and release $r2$ was accompanied by an increase in duplication that was even more pronounced.

After some further investigation the underlying cause was identified: from version $r1$ to $r2$ a particular piece of hardware was upgraded for some installations of the system. As a result, the driver for that hardware needed to be changed as well. Since the old driver was still needed for the old piece of hardware, the driver was copied completely, and a few minor changes were made to it to facilitate the new hardware.

Although there may have been a reason for copying initially

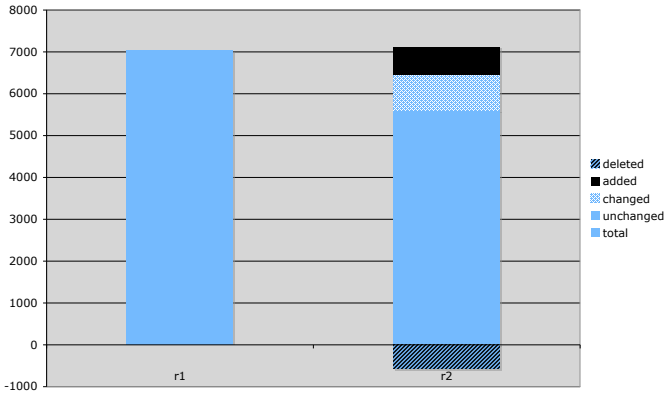


Fig. 8. Code modifications on the file level, between two versions (Case V-B).

(shorter time to market, a belief that this was a temporary solution), it will be a burden to maintenance cost in the long run. As the charts show, the initial copying was not subsequently cleaned up in later versions of the system.

The fundamental problem, of course, was that no explicit decision was ever made: the developers took this course of action because they felt this was the best, easiest, or quickest way to satisfy a particular change request, but they did not take into account the consequences to the overall system health.

By analysing the numbers, and showing the result of the action on the complete code base, we provide both developers and management with the data needed to make an informed, explicit decision regarding the future of the system. In this case, actions were planned in subsequent releases to curb system erosion by refactoring the code such that the introduced duplication was eliminated again.

B. Systems accounting from code churn

An organisation was building an information system to automate its primary business process. System development was done on three sites on two continents. Requirements gathering and management was done on another two sites in two different countries. An estimated 25 subcontractors were involved in the project, with an estimated 240 people. The perception of higher management was that a significant part of the 240 people involved were developing code on a day-to-day basis. We were asked to monitor the technical quality of the system.

When we analysed the system, it turned out to be quite large. It was developed in a modern object-oriented language, and consisted of about 1.5 million lines of code divide over about 7000 files. Based on software productivity statistics [7], a system of this volume, built with this technology, corresponds to about 170 man years, as a rough estimate. The technical quality of the system, judged on the basis of indicators such as modularization, complexity, duplication, etc. did not reveal any major risks, though several points for improvement were identified.

Since the system was several years old already, strong doubt arose whether for a system of this size and quality the staffing of 240 people was justified. To answer this question from management we made an in-depth comparison of two versions of the system, separated by about one year.

The overall volume of the system, its complexity, duplication, and other indicators turned out to have been more or less stable over that period. Still, many modifications had been made. We charted those modifications in terms of file creations, deletions, and modification, as can be seen in Fig. 8. When a file was copied, the copy was adapted, and the old file was removed, we counted this as a single file modification.

Based on our measurements, it turned out that the amount of change over that year was nowhere near the productivity that one may expect from a 240 people effort. Perhaps 50 would have been more realistic.

After we reported our findings, an investigation was started to establish how many staff members were actually active as software developers, to find out what exactly the other people on the project were doing, and what sources of overhead could be eliminated. This investigation led to a restart of the project with less than 30 people, of which about 18 are actively developing software. Development of the system has been brought back to a single location. Requirements gathering is still done at a different location, but people responsible for the requirements spend at least three days a week at the development location.

After the restart, the productivity of the project grew *in absolute terms*. We were told that the project was delivering more functionality (defined in terms of feature requests or change requests) per time unit with 30 people than they were with 240 people.

In retrospect this is not as surprising as it seems. It is widely acknowledged that adding more manpower to a software project does not make it necessarily more productive [8]. In addition, dividing resources over a multitude of locations was identified as a major source of overhead and waste.

What was surprising to us is that our technology apparently can be used for what we call ‘systems accounting’. Using a very technical (and not very sophisticated) measure, we were able to see right through the 25 subcontractors and the 240 people. In this case, software monitoring at the system level and fact-based interaction with high-level management proved to be decisive in radically improving efficiency.

C. Learn from failure, measure for success

We were asked to monitor the development of an e-Commerce platform for a large bank in the Netherlands. This platform was built from scratch as a replacement for a functionally identical platform which failed in the rollout phase. Because of the earlier failure, the bank and its (new) outsource party decided to use our monitoring service in order to gain insight into the technical quality of the new software. Their key targets for this new project were to realize low costs of maintenance and operation.

At the start of the second attempt, we performed an assessment on the code base of the first attempt, which led us to attribute the failure to several factors, including:

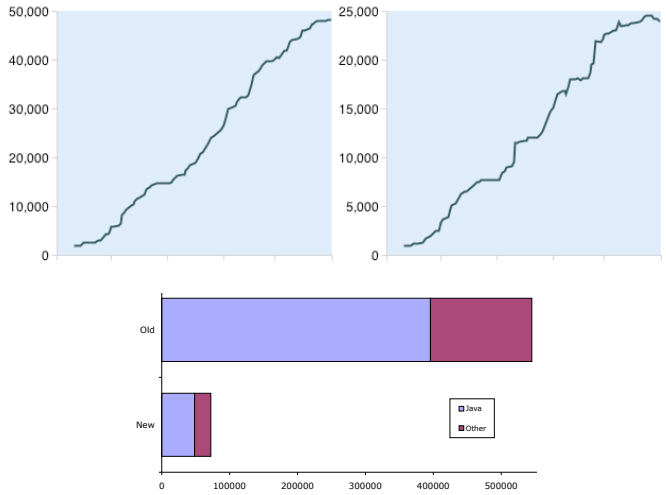


Fig. 9. Volume measurements in lines of code for both the failed project and the new project (Case V-C).

- An overly complex design that ambitiously tried to generalize the e-Commerce platform so that it could handle future markets.
- A design that tried to solve possible performance issues a priori by using a complex distributed J2EE EJB-based architecture.
- A lack of (automatic) unit tests and integration tests.

Learning from these lessons, a set of contrasting goals were set for the new project:

- A minimalist design that focused only on the current product line.
- A lightweight approach to system architecture.
- A test-driven approach, by focusing on automated system and unit tests.

Continuous monitoring of the source code base was put into place from the start of the new project.

Our monitoring of volume indicators demonstrated that the new approach resulted in a much leaner system. Fig. 9 shows measurements of the lines of code for both the old and new system. The measurements are split out between Java code and other kinds of code, which include HTML, XML, and SQL. As the charts show, the new system was significantly smaller than the old one (about 7,5 times smaller). Over the period of 14 months, the volume increase of the new system was almost linear, indicating constant productivity throughout that period.

The new approach also paid off demonstrably in terms of quality indicators such as complexity and duplication. The complexity profiles of both old and new system are shown in Fig. 10. Using the quality model of Section IV, the old system is rated on complexity as poor (--), while the new system is rated as excellent (++). The duplication measurements are shown in Fig. 11. The new system contains higher duplication in non-Java code (33%) than in Java code (2%), but significantly less than the old system for both kinds of code (23% and 57%). The timelines reveal that at the start of the project, duplication was low and relatively unstable,

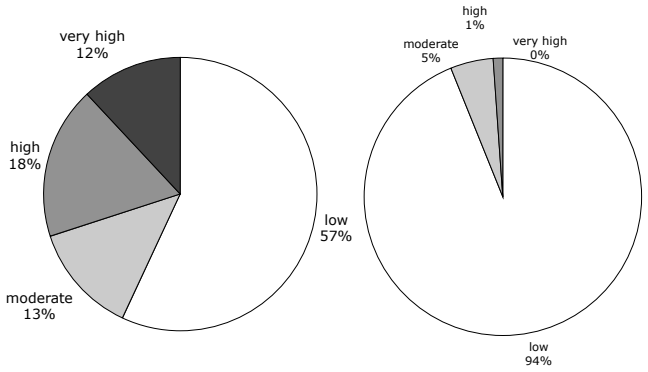


Fig. 10. Complexity profiles for both the failed project and the new project (Case V-C). The former system scores --, while the new system scores ++.

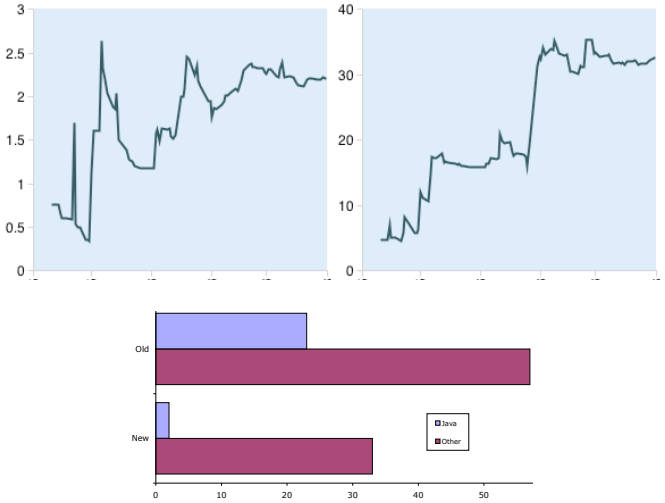


Fig. 11. Duplication measurements in lines of code for both the failed project and the new project (Case V-C).

while towards the end of the project, this measure stabilizes. For Java code, the final value is excellent, but for non-Java code duplication is still too high.

We also measured several coding standards, including:

- Double checked locking: 60 instances in the old system and 0 in the new. The double checked locking construct introduces a bug in thread synchronization code.
- String reference equality: 22 vs. 0. String reference equality is often a bug introduced by inexperienced programmers; the equals method should be called instead.
- Too generic error handling constructs: 2141 vs. 17. The 17 cases in the new code were manually checked, and did not introduce risks (false positives), while sampling the 2141 violations in the old code revealed actual problems.

Monitoring also revealed that test code was being written, with a test coverage stable at about 60% throughout the course of the project.

In this case, monitoring helped to reduce the size and increase the technical quality of the new system. Surprisingly, the much simpler (non-distributed) architecture of the new system performed much better than the original. Also the

resulting system proved to be much easier to tweak for specific performance issues. In contrast to the previous attempt, this system was successfully concluded and taken into production.

VI. Conclusion

A. Contributions

Some years ago, we developed our tool-based method for software monitoring [1] and introduced it into the market. Since then, we have applied the method in a wide range of circumstances. On the technological side, we have monitored systems built in modern object-oriented languages, as well as classical mainframe languages. On the organizational side, we have acted on behalf of both clients and providers of application outsourcing services (but never both at the same time, naturally). In terms of software life-cycle, we have monitored both system maintenance and development from scratch.

More recently, we have used our experience in software monitoring as well as in software risk assessment [9] to draft a practical model for measuring the technical quality of software products [5]. This model has been instrumental for the abstraction and aggregation of source code analysis results necessary for translation of technical findings into management-level notions.

In this paper, we have summarized both the monitoring approach and the quality model, and presented them for the first time in combination. Moreover, we have shared our experience of applying the approach and the model in a range of application outsourcing situations. The cases reported include maintenance as well as development situations, various technology mixes, and a variety of organizational contexts.

B. Lessons learned

Among the lessons learned from these cases and others not reported here, are the following:

- Simple measures, but well-chosen and aggregated in meaningful ways, are effective instruments for software monitoring.
- The simultaneous use of distinct metrics can be used to zoom in on root causes of perceived quality or productivity problems.
- The gap between technical and managerial realities can be bridged with a practical software product quality model.
- Monitoring helps to curb system erosion in maintenance situations.
- Monitoring code churn allows 'systems accounting'.
- Monitoring helps to achieve clear productivity and quality targets.
- The chances of success of software development projects are influenced positively by software monitoring.

In the introduction, we indicated that software monitoring, when executed by a third party, can resolve a dilemma that arises from application outsourcing. Indeed, the paradox of removing technical know-how from the organization to an outsourcing party while needed that knowledge to manage the relationship to that party, can in our experience be solved by third-party monitoring of the technical quality of the outsourced software.

C. Future work

Software monitoring, though supported by tools, standards, and models, is a creative process that needs continuous improvement and calibration. Our quality model is still fairly young, and will be further refined based on our experience with its application. Also, changes in the maturity of the software industry will call for adaptation of rating guidelines, application of more sophisticated metrics, and perhaps other analysis instruments. In particular, we are keeping a close watch on developments in the area of analyzing and visualizing software evolution.

We collect a extensive set of measurement data in the course of our monitoring and assessment activities. We are currently consolidating this data into a benchmarking data base that will allow well-founded comparisons of systems on the level of source code properties as well as system-level quality aspects.

References

[1] T. Kuipers and J. Visser, "A tool-based methodology for software portfolio monitoring." in *Proceedings of the 1st International Workshop on Software Audit and Metrics, SAM 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, M. Piattini and M. Serrano, Eds. INSTICC Press, 2004, pp. 118–128.

[2] ISO, *ISO/IEC 9126-1: Software Engineering - Product Quality - Part 1: Quality Model*. Geneva, Switzerland: International Organization for Standardization, 2001.

[3] ——, "ISO/IEC TR 9126-2: Software engineering - product quality - part 2: External metrics," Geneva, Switzerland, 2003.

[4] ——, "ISO/IEC TR 9126-3: Software engineering - product quality - part 3: Internal metrics," Geneva, Switzerland, 2003.

[5] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring mainainability," 2007, draft, April 30.

[6] C. M. Software Engineering Institute, "Cyclomatic complexity – software technology roadmap," http://www.sei.cmu.edu/str/descriptions/cyclomatic.html.

[7] Software Productivity Research LCC, "Programming Languages Table," Feb. 2006, version 2006b.

[8] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing, 1975.

[9] A. van Deursen and T. Kuipers, "Source-based software risk assessment," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 385.

MAIS: an awareness mechanism for change identification on shared models

Marco A. de M. Lopes[1], Cláudia M. L. Werner[1], Marco A. S. Mangan[2]
[1] *System Engineering and Computer Science*
COPPE – Federal University of Rio de Janeiro
P.O Box 68511 – CEP. 21945-970 – Rio de Janeiro – RJ – Brazil
[2]*School of Computer Science – Pontifical Catholic University of Rio Grande do Sul*
Porto Alegre – RS - Brazil
{mlopes,werner}@cos.ufrj.br, mangan@pucrs.br

Abstract

Awareness mechanisms could reduce the isolation among distributed software development teams. In particular, they can be applied to concurrent modeling of software artifacts, where a software development team needs to keep track of the evolution of shared models. The developer's conception of a shared model can be continuously updated with this kind of mechanism. This paper presents an awareness mechanism that collects artifact change information directly from the developer workspace, not influencing his workflow. The objective of this mechanism is to help developers to perceive concurrent artifact changes and coordinate actions to minimize the effort of getting a consistent global state of the shared artifact. Change information is classified, grouped and filtered to reduce a possible cognitive overload. An observation study was performed, aiming to infer some indicators about the utility of the mechanism.

1. Introduction

Due to businesses globalization, organizations had to rethink and reevaluate their structures and procedures, to remain competitive in the market. Software organizations are also affected by this trend. These organizations often search for external solutions (e.g. outsourcing) in different locations, to explore the advantages offered in those places.

Global Software Development (GSD) [13] takes into account technical, social, and economical aspects of developing software in a distributed setting. This distribution is either geographical (members of a software development team are dispersed spatially) or over time (team members collaborate in alternated schedules).

Herbsleb and Moitra [13] present some factors that motivate GSD: (i) need to obtain scarce resources with certain profiles; (ii) proximity to the software consumer market; (iii) fast formation of corporations and virtual teams, to explore business chances; and (iv) pressure to provide time-to-market, exploring the possibility of increasing productive work hours using the hourly spindle differences among software development team members. These factors are intrinsically related to the increase of productivity and the reduction of costs on software development. This "virtualization" of teams and organizations leads to some difficulties on the interaction of team members.

Despite these factors, there are tasks throughout the software lifecycle that still require interactions among individuals: creative brainstorming, pair-programming, and peer review are some tasks that are usually performed by more than one individual. In the context of a distributed software development organization, the enactment of this kind of task needs to be adapted in a way that supplies the lack of some aspects found in a face-to-face interaction (such as communication easiness).

The software modeling activity is a software development task that can be done in a distributed setting. For example, delays in schedule could force the adoption of task parallelism to meet the project deadline. In such case, the modeling activity is divided among different developers. More than one developer can develop different model views, exploring parallelism among tasks.

Developers can interact with each other through the work (contributions) on individual copies of a shared software model. From time to time, these copies must be synchronized with a copy stored in a central repository [24]. The divergence among contributions over the shared model can cause a convergence effort, often culminating in rework.

In Computer Supported Cooperative Work (CSCW), an awareness mechanism for shared workspaces is an alternative to assist tasks that present some characteristics of the previously described scenario. They provide information to developers about which changes have been done, by who, when, where and how they were done [12]. In the case of interactions over copies of a shared software model, change information is collected and presented to speed up the detection of possible conflicts between the copies of the developers. Therefore, change information makes coordination of actions possible. This kind of mechanism can be found in some general groupware, but groupware task support is not as complete as a specific single-user tool [16]. Sometimes users have to change their way of work to use this kind of mechanism found in general purpose groupware.

Figure 1 presents a scenario where an awareness mechanism can be applied. Developers collaboratively design a software model ('M'), at different time and location. It is assumed that developers D1 and D2 work simultaneously on their copies in [T1, T2] period of time. Now, suppose that developer D3 modifies 'M' in a T0 instant of time, before any contribution has been made by D1 and D2. When D1 and D2 begin to contribute over their copies of 'M', they must be aware of the changes made by D3. Also, they have to be aware of each other's contributions, as well as who did them. If D3 contributes again some times later (T3), he must be aware of D1 and D2 changes. Thus, an awareness mechanism is useful to make this change information available to all developers.

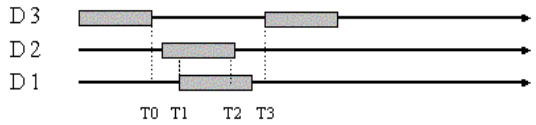


Figure 1. Interactions over a shared model

This paper presents a Multi-synchronous Awareness Infra-Structure (MAIS) mechanism and prototype that aim to collect, distribute and present change information over shared software models. This information is offered to developers who participate in a collaborative modeling interaction. MAIS was developed to be integrated to an existing environment or editor. The change information related to the state of copies of the shared model is filtered and organized for its presentation. It helps developers to understand changes made over the other copies of the shared model. The proposed approach aims to offer this kind of support without breaking down the developer's workflow, registering the developer`s awareness state in the shared group memory. The current version of MAIS prototype considers changes over UML models (in particular, class models) developed on the OdysseyShare Environment [21].

The remainder of the paper is organized as follows. The Background work is presented in Section 2. The design of the awareness mechanisms MAIS is discussed in Section 3, and its prototype is presented in Section 4. An analysis of a preliminary viability study is discussed in Section 5. Finally, Section 6 provides the conclusion and future work.

2. Background Work

In the GSD context, developers can deal with two sources of complexity [9]: (i) the development complexity of a software artifact, and (ii) the complexity of the distributed software development process itself. So, it is important that developers have some kind of tool support for executing their tasks.

CSCW provides many definitions to the awareness conceptIn this paper, awareness means an understanding of the activities of others. This provides a context for a developer's own activity. This context is used to ensure that individual contributions are relevant to the group activity as a whole, and to evaluate individual actions with respect to the group goals and progress. This information allows groups to manage the process of collaborative working [7]. It is the understanding of a system state (common artifacts where individuals work on, including a particular representation of these), as well as past activities, current state and future options [26]. According to Gutwin and Greenberg [12], the information used to reach an awareness state could be obtained through: (i) transference of others activities information about collaboration; (ii) transference of the information generated by the contributions over some shared artifacts; and (iii) intentional communication between collaboration members.

Rosa *et al.* [23] suggest that the awareness state can be influenced by the context where the collaboration occurs. The elements of this context must be identified and represented to increase the awareness state of collaborators. They consider that these elements can be grouped into five categories of information, related to:

(i) individuals and groups; (ii) tasks previously established; (iii) the relationship between people and tasks; (iv) the environment where collaboration happens; and (v) concluded tasks. This taxonomy was proposed to help the identification of context elements and the implementation of mechanisms that collect and distribute these elements to collaboration individuals.

Awareness mechanisms are proposed aiming to offer some indications about what happened, when, how, where, and who has done something. They intend to balance the amount of information that needs to be presented [2]. These mechanisms are implemented to help collaborators on the way they interact, which can be classified as [20]: (i) synchronous, at the same time using the same data; (ii) asynchronous, using the same data but not necessarily at the same time; and (iii) multi-synchronous, where each collaborator has a copy of the shared data and, at certain periods of time, synchronizes his copy to get an updated view of the shared data.

To have a complete notion of a collaborative interaction to which they belong, the individuals have to be capable to realize: (i) the social context of groups where they are members, (ii) the context of the activities that they participate, and (iii) changes made on their workspace during the interactions [2].

This approach uses the shared workspace to obtain the awareness information, which is the immediate understanding of actions of individuals [12]. It involves knowing where others are working, what they are doing and what they will probably do.

Tam *et al.* [28] describe the change awareness concept as the ability of individuals to recognize changes done in a collaborative artifact by another participant. Keeping the change history helps the collaboration members to remember past actions, and contextualize new members about the progress of the current activity.

Some approaches are found in literature to aid a collaborative software modeling task. CO2DE tool [19] is an implementation of a graphical editor of UML diagrams. It is based on a "masks" metaphor, which represents diagram versions. Another UML collaborative editor is D-UML [3]. Tukan [25] is a distributed environment for Smalltalk programming. SAMS environment [20] allows collaborative edition by synchronous, asynchronous and multi-synchronous modes of interactions. NetEdit [31] is a collaborative web-based text document editor.

There are collaborative tools that are non-obtrusive while collecting and distributing awareness information, that is, these actions do not disturb developers when they are doing a collaborative task. Palantír tool [24] complements configuration management systems by offering information about workspaces of other developers in a collaborative session. Kobylinski *et al*. [15] present an approach of an awareness system that allows collaborators to monitor activities of others over software artifacts.

These approaches can be classified according to the interaction mode between individuals. For example, CO2DE, D-UML, and Tukan approaches offer support to synchronous interaction (CO2DE also provides asynchronous support). Palantír, SAMS, NetEdit and the approach described in [15] work in a multi-synchronous way, providing support to synchronous and asynchronous modes too.

However, these approaches do not present the developer's awareness of a given change over shared artifacts as the approach proposed in this paper does. This information can be useful, since it prevents information overload to the "aware" developer.

Some studies [28] analyze the graphical representations that illustrate changes over UML class diagrams, besides presenting results of an empirical study that determines strong and weak points of these representations. These studies indicate that textual change representations (who changes, what is changed, when it changes) are the ones that have greater impact among developers. The MAIS approach uses textual change representation, as described in the next section.

3. Design

MAIS approach involves an awareness mechanism that uses changes made on shared software models as awareness information. It collects, distributes and presents this information to developers that are interacting on a shared software model.

By using MAIS, developers interact in a multi-synchronous mode to manipulate shared software models, offering to developers an independent way of working. The contributions over a shared model are done by each developer's local copy. So, it is possible to manipulate this artifact in a concurrent way. MAIS mechanism is meant to be used before the convergence phase [29] of the shared model copies into a global one, stored in a central repository.

The individual contributions over the shared model are propagated to all developers, offering an overall view of what is being done, which can be useful to make a global “conscience” of the shared model. This can help developers on the convergence phase, since conflicts can be avoided by using this kind of information. We assume that the convergence of the

shared model copies occurs when developers wish to check-in their changes.

The MAIS specification is not dependent of a modeling tool; it is a reusable software component [11] that can be coupled to various CASE editors or software environments. However these tools must satisfy some basic extensibility requirements, for instance, to provide an extensibility API. The modeling tool must also have a notification mechanism about fine grained model changes.

Contributions over the shared model generate the awareness information, which are represented using an event metaphor. It brings the idea of event notification systems [22] [8] [6], where developers register themselves on the collaboration to be notified when a certain event (some contribution over the shared model) occurs. For each event occurrence, this awareness information is distributed to all developers that are registered.

The awareness information is collected using a sensor mechanism [22], which is associated to some particular model elements. These sensors are installed on the modeling tool and capture change information, forwarding them to the MAIS mechanism that broadcasts it.

The event concept is implemented using the 5W+1H concept: an event is described by an action (How) over a shared model (Where) element (What). These actions are done by a developer (Who) in a certain period of time (When). MAIS events also describe relevancy relations; events related to changes in composite elements are grouped in a single event.

Grouping Events

The change information can be manipulated to highlight some “hidden” characteristics about the interactions over the shared model. It is useful to help developers on doing their model contributions, because this enriches their knowledge about the context where the interaction occurs. Grouping events by a characteristic (for example, by developer) makes it possible to observe where actions (represented by events) are located over the shared model. For example, let us consider a UML class model as the shared model presented in Figure 1. Grouping change information of a specific UML class by developer can lead us to infer how much each developer knows about each class, i.e., the volume of changes can indicate the developer’s knowledge about the class.

Ranking Events

Apart from grouping events by some characteristics, it is interesting that developers are provided with information about events that are important to their work. An event ranking can be established with this goal. A value can be attributed to each event to determine its importance for a given developer.

Thus, MAIS introduces the event relevance concept. It aims to measure how important some kind of event is to a developer by analyzing other past events.

This value, called Relevance degree (Gr), related to the event "e" for "D", is calculated by applying the following formula:

$$Gr(e,D) = Er(e,D)/Eg(D)*100$$

where "Er" represents the amount of events that a developer "D" generated, relative to a shared model element "e"; and "Eg" is the total amount of events generated for "D".

For example, in a shared UML class model, if a "D1" developer has generated 40 change events, being 5 of these related to the "C" class, events which involve the "C" class (element) have a relevance degree of 12.5% for "D1".

Filtering Events

Finally, developers do not want to spend too much time searching for relevant events, since their work flow can be negatively influenced by long duration searches. The grouping and relevance concepts should help to reduce the number of events that each developer has to analyze.

However, the amount of events grouped by category can become very large. It is reasonable to imagine that many of these events may have been noticed by developers during the session and it might not be productive to show them again. Thus, event filtering can attenuate the overload problem.

MAIS mechanism does event filtering driven by a certain criteria, which is defined at the design phase of the mechanism implementation. The event aware information concept indicates the event existence information noticed by a developer. As such, events can be marked as "aware". This developer's "aware state" can be used to infer the knowledge of others about the contributions (I know that you know/don't know) [5].

4. Implementation

MAIS prototype was implemented as part of the OdysseyShare Environment [21], using the Java

platform. The prototype uses a tuple space [4], which stores the generated events. This tuple space works as a shared workspace. The prototype uses UML class model as the shared software model.

Figure 2 presents MAIS event model. The event concept implementation consists of an action ("Event" abstract class) combined with a model element ("Element" abstract class). The supported actions are: (i) register itself on the interaction ("Register" class); (ii) create ("Create" class); (iii) delete ("Delete" class); and (iv) modify ("Update" class) a model element. When an element is modified, "Change" class represents the changes over this model element. The supported UML class model elements are: (a) class, (b) attribute, (c) method, and (d) relationship.

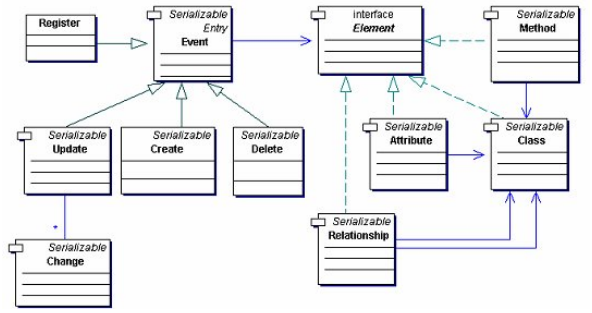


Figure 2. MAIS event model

The tuple space works as a repository of events; objects are stored, removed and searched in this space. This is installed in some network node, accessible to all developers through the mechanism. When an interaction over the shared model begins, the developers register themselves on the mechanism, to be notified when the objects are stored in the tuple space.

GigaSpaces server [10] is the used implementation of JavaSpaces specification [14], which is an application of the tuple space concept. When new events are created, developers are notified and are able to retrieve them from the tuple space.

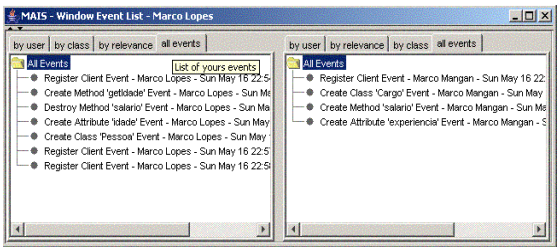


Figure 3. Presentation of generated events

As mentioned in the previous section, MAIS prototype is coupled to a software development environment or CASE tools in the following way: the environment/tool must provide some API that reflects the supported changes on a certain shared model (for example, an element creation) and it is necessary to register MAIS to listen to these events. Thus, when a change is made on the shared model, the environment/tool notifies MAIS about this event.

The events are presented to developers as text messages, describing the event that generated it, and the model elements affected by the event. The events are presented in two lists, depending on who generated them. Events generated by the current developer are presented in the left hand side list, while events generated by others are presented in the right hand side list (Figure 3).

5. Observation Study

Since the awareness concept is associated to an individual's mental state [26], it is important to observe the usage of MAIS prototype. This observation can provide information about the mechanism applicability in a given scenario, identifying deficiencies and new requirements. This section presents a qualitative observation study about MAIS prototype use [30]. This empirical study consists on observing its use within a collaborative modeling session, registering data related to aspects such as user's satisfaction on using the prototype, improvements suggested by users, and so on.

This preliminary study does not aim to test hypothesis; we aim to perform a exploratory study. It is assumed that all developers are working on the same version of a shared model. It is important to highlight that it is not a prototype goal to generate a consistent global state of the shared model. It only provides the information about generated changes, which is used by developers to guide their changes over the model copies.

All MAIS users have the same role (i.e., developer). There is no coordinator/moderator role; individuals make contributions over the shared model trying to generate a consistent version, discussing and deciding how changes can be implemented.

The empirical study's objective is described below, following the structure presented in [30]:

Analyze the MAIS mechanism in an informal concurrent modelling activity, related to changes made in a shared model.

Intending to characterize the mechanism usefulness, referring to users' satisfaction in using it and task applicability.

Regarding satisfaction with the performance and efficiency in modeling activities.

In the developer's point of view.
Within the OdysseyShare environment context.

For the study enactement, ten volunteers were invited. Four of them are computer science undergraduate students at the Federal University of Rio de Janeiro (UFRJ). The other six are Systems Engineering and Computer Science graduate students at UFRJ. Only one volunteer had previous experience in using OdysseyShare environment. All volunteers have some experience in object-oriented modeling. However, no volunteer had previous experience with the MAIS prototype.

The instruments used to execute the study were adapted from [17] and involve: (i) a commitment term; (ii) two questionnaires, to be answered by participants; and (iii) a document describing the task, distributed to each participant. All volunteers used OdysseyShare environment (v.1.6.0) to execute the task. Four volunteers were selected to use this environment version combined with the MAIS prototype (v. 1.1.0). Two volunteers' sessions were used to calibrate the study execution.

The following steps summarizes study: (i) each participant signs the commitment document; (ii) they answer the characterization questionnaire; (iii) they read the task description; (iv) if necessary, they listen to an oral explanation of MAIS prototype or OdysseyShare usage; (v) they accomplish the task, and (vi) they fill in the task accomplishment questionnaire.

The study task consists of a concurrent software modeling session of a UML class model, where two developers interact over the model at the same time. Each one has some activities to do, and when they finish, a meeting is established to merge their contributions. The document that describes the task presents a brief description of the model elements. The used model was derived from MAIS event representation class model.

Two groups of sessions were organized: (i) one with developers that used the prototype, and (ii) a control group without the prototype support. Each pair of developers received the same version of the shared UML class model. In both cases, the changes were made using OdysseyShare environment.

Each participant reads the instructions described in the task description document. A verbal explanation about the prototype purpose is done for those who use it. Participants act as developers, interacting concurrently over the shared class model. Changes are registered using the event metaphor, as previously described.

Preliminary Analysis

After group (i) session (i.e., MAIS users), an informal meeting was taken to discuss some general points related to the prototype. All volunteers indicated that the prototype should notify them when an important event occurs. They suggested that an important event should call the user's attention (e.g., blink on the computer screen). This characteristic would remind developers to turn to MAIS interface to observe the shared model evolution. They found the approach very useful, in the sense of situating the possible "conflict areas" on the shared model, concentrating their attention to the classes that they are working with.

The same procedure (i.e, an informal meeting after study execution) was taken to the group that didn't use the MAIS prototype. Due to the task simplicity, the volunteers didn't externalize the need of a tool to support the task execution. The generated conflicts were reasonable treated by using the OdysseyShare environment alone.

Based on the data collected during the informal meetings, the study has demonstrated some points for improvements regarding the prototype: (i) the critical changes for some developer must be highlighted, possibly by playing a sound alarm or blinking on the computer screen as suggested by some participants; (ii) the questions must be revised, in terms of bias that was detected in some points. For example, when the volunteers answer the questions related to the proposed task, they unintentionally give high grades for them. Finnally, (iii) the task should be improved to provide more conflicts betweens copies of the shared model. The number of occurrences and the complexity of the presented conficts turned out to be easy for the execution of the task with or without the MAIS support.

6. Conclusion

This paper described the isolation problem among distributed developers that share software models, in particular, UML-based models. In the CSCW research field, the awareness concept is presented as the basis for a technique that is meant to reduce this isolation. To make it possible, change state information related to a shared model is propagated through awareness mechanisms to developers who interact over it.

Awareness support is present in groupware tools. However, the task support offered by general groupware is not as complete as the one found in similar single-user applications [16]. Changing a

single-user modeling tool for an equivalent groupware can cause a negative effect in the productivity and the satisfaction of developers. Also, the learning curve for using new tools can be inadequate to the team‘s expectation.

This paper proposed an awareness mechanism for change identification on shared models, and a corresponding prototype named MAIS. Both were developed to be general purpose, not being restricted to a specific modeling tool. The kind of models supported by the mechanism is represented using UML notation.

The MAIS awareness mechanism monitors contributions made in the local workspace of each user. It is non-obtrusive, using an event metaphor [22] to propagate this information to every developer involved. The information overload characteristic is considered when change information is presented. MAIS mechanism is able to identify local changes that were not yet committed in the central version control repository. The developers are notified in cases of potential modeling conflicts, enabling preventive actions [24].

Three procedures are considered by MAIS mechanism to present the shared model evolution information: (i) classification, (ii) grouping, and (iii) filtering of events. Filters are established to reduce information overload related to changes on shared software model. The mailbox metaphor is applied in this procedure. Events already processed by the end-user are presented in a different area of the user interface. Another concept adopted in filtering is the “awareness of awareness”. The event processing information can be accessed by the other developers. New actions can be driven according to this kind of information.

One of the main contributions of this paper is the design and implementation of an independent and non-obtrusive mechanism that collects and distributes change information related to copies of a shared software model. The collected events are available to be used by other tools, being possible to do some data analysis outside the mechanism. The use of software models allows to detect inconsistencies in initial phases of the software lifecycle, especially when these models are described in a well-formed language such as UML. The relevance and change aware information is obtained in the interaction process over a shared model, not impacting the developer's workflow.

The observation study highlights the importance of notifing developers about changes. This can be done by using some kind of alarm. This feature was reported by all developers, because sometimes they “forget” the existence of the tool. MAIS prototype behaves as expected, offering a global notion of contributions over the shared model and identifying the critical “pieces” of the shared model that each developer works on.

As future work, we consider the need of exploring new ways of awareness information presentation, representing the evolution of the shared model. There is the need to complete our work by searching for some patterns of change information that could be useful on the detection of conflicts and decision making. These patterns can help the identification of a non explicit ability of a team member and provision of information about productivity and quality of the produced artifacts. Some kind of automation for conflicts detection needs to be designed and implemented. A case study must be applied in an industrial setting. We also realized that it is necessary to review the proposed task in order to have more conflicts situations, to observe the prototype behavior in this context. A quantitative approach will be adopted in a next version of this study, and training on the modeling tool will be prepared.

7. References

[1] ALTMANN, J., POMBERGER, G., 1999, "Cooperative Software Development: Concepts, Model and Tools" Proceedings of the Technology of Object-Oriented Languages and Systems, pp. 194-209, Santa Barbara, California, United States, August.

[2] ARAÚJO, R. M., 2000, “Extending the Software Process Culture - A Groupware and Workflow-Based Approach”, DSc. Thesis, COPPE-UFRJ, Rio de Janeiro, RJ, Brazil.

[3] BOULILA, N., DUTOIT, A. H., BRUEGGE, B., 2003, "D-Meeting: an Object-Oriented Framework for Supporting Distributed Modeling of Software" International Workshop on Global Software Development, International Conference on Software Engineering, pp. 34-38, Portland, Oregon, United States, May.

[4] CARRIERO, N., GELERNTER, D., 1989, "Linda in context", Communications of ACM, v. 32, n. 4, pp. 444-458.

[5] DAVID, J. M. N., BORGES, M. R. S., 2001, "Selectivity of Awareness Components in Asynchronous CSCW Environments". In: Proceedings of 7th International Workshop on Groupware (CRIWG 2001), pp. 115-124, Darmstadt, Germany, September.

[6] DE SOUZA, C. R. B., BASAVESWARA, S. D., REDMILES, D. F., 2002, "Supporting Global Software Development with Event Notification Servers". In: Proceedings of 24th International Conference on Software Engineering, International Workshop on Global Software Development, pp. 9-13, Orlando, Florida, United States, May.

[7] DOURISH, P., BELLOTTI, V., 1992, "Awareness and coordination in shared workspaces" Proceedings of the 1992 ACM conference on Computer-supported cooperative work, pp. 107-114, Toronto, Ontario, Canada.

[8] FARSHCHIAN, B. A., 2001, "Integrating geographically distributed development teams through increased product awareness", Information Systems., v. 26, n. 3, pp. 123-141

[9] FROEHLICH, J., DOURISH, P., 2004, "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams". In: Proceedings of 26th International Conference on Software Engineering, pp. 387-396, Edinburgh, United Kingdom, May.

[10] GIGASPACES, 2006, "Gigaspaces Grid Server". In: http://www.gigaspaces.com/docs/doc/index.htm, Accessed in 22/04/2006.

[11] GRUNDY, J., HOSKING, J., 2002, "Engineering plug-in software components to support collaborative work", Software: Practice and Experience, v. 32, n. 10, pp. 983-1013

[12] GUTWIN, C., GREENBERG, S., 2001, "A Descriptive Framework of Workspace Awareness for Real-Time Groupware", Computer Supported Cooperative Work, v. 11, pp. 411-446, Kluwer Academic Publishers.

[13] HERBSLEB, J. D., MOITRA, D., 2001, "Guest Editors' Introduction: Global Software Development,", IEEE Soft., v. 18, n. 2, pp. 16-20, IEEE Computer Society Press.

[14] JAVASPACES, 2006, In: http://java.sun.com/developer/Books/JavaSpaces. Accessed in 22/04/2006.

[15] KOBYLINSKI, R., CREIGHTON, O., DUTOIT, A., et al., 2002, "Building awareness in global software engineering: using issues as context" International Workshop on Global Software Development, pp. 18-22, Orlando, Florida, United States.

[16] LI, D., LI, R., 2002, "Transparent sharing and interoperation of heterogeneous singleuser applications". In: Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work, pp. 246-255, New Orleans, Louisiana, United States.

[17] MANGAN, M. A. S., ARAÚJO, R. M., KALINOWSKI, M., et al., 2002, "Towards the Evaluation of Awareness Information Support Applied to Peer Reviews". In: Proceedings of 7th Conference on Computer Supported Cooperative Work in Design (CSCWD´2002), pp. 49-54, Rio de Janeiro, Brazil, September.

[18] MANGAN, M. A. S., BORGES, M. R. S., WERNER, C. M. L., 2004, "Increasing Awareness in Distributed Software Development Workspaces". In: Proceedings of 10th Groupware: Design, Implementation, and Use, v. 3198, pp. 84-91, San Carlos, Costa Rica, September.

[19] MEIRE, A., BORGES, M. R. S., ARAÚJO, R. M., 2003, "Supporting Collaborative Drawing with the Mask Versioning Mechanism". In: Proceedings of 9th Groupware: Design, Implementation, and Use (CRIWG 2003), v. 2806, pp. 208- 223, Autrans, France, September.

[20] MOLLI, P., SKAFA-MOLLI, H., OSTER, G., et al., 2002, "SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments". In: Proceedings of 7th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2002), pp. 80-84, Rio de Janeiro, RJ, Brazil, October.

[21] ODYSSEY, 2006, "OdysseyShare Project". In: http://reuse.cos.ufrj.br/odyssey, Accessed in 22/04/2006.

[22] PRINZ, W., 1999, "NESSIE: an awareness environment for cooperative settings". In: Proceedings of the 6th European Conference on Computer Supported Cooperative Work, pp. 391-410, Copenghagen Denmark.

[23] ROSA, M. G. P., BORGES, M. R. S., SANTORO, F. M., 2003, "A Conceptual Framework for Analyzing the Use of Context in Groupware". In: Proceedings of 9th Groupware: Design, Implementation, and Use, v. 2806, pp. 300-313, Autrans, France, October.

[24] SARMA, A., NOROOZI, Z., VAN DER HOEK, A., 2003, "Palantír: raising awareness among configuration management workspaces". In: Proceedings of 25th International Conference on Software Engineering (ICSE), pp. 444-454, Portland, Oregon, United States, May.

[25] SCHÜMMER, T., SCHÜMMER, J., 2001, "Tools for XP Development - Support for Distributed Teams in eXtreme Programming". In Succi, G. and Marchesi, M.,eXtreme Programming Examined, Addison Wesley.

[26] SOHLENKAMP, M., 1998, Supporting Group Awareness in Multi-User Environments through Perceptualization, Msc Dissertation - Fachbereich Mathematik, Informatik der Universität - Gesamthochschule - Paderborn.

[27] TAM, J., GREENBERG, S., 2004, "A Framework for Asynchronous Change Awareness in Collaboratively-Constructed Documents". In: Proceedings of 10th Groupware: Design, Implementation, and Use, v. 3198, pp. 67-83, San Carlos, Costa Rica, September.

[28] TAM, J., MCCAFFREY, L., GREENBERG, S., 2000, Change Awareness in Software Engineering Using Two Dimensional Graphical Design and Development Tools. Report 2000-670-22, Department of Computer Science, University of Calgary, Alberta, Canada.

[29] VIDOT, N., CART, M., FERRIÉ, J., et al., 2000, "Copies convergence in a distributed real-time collaborative environment". In: Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work, pp. 171-180,Philadelphia, Pennsylvania, United States.

[30] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M.C., REGNELL, B., WESSLÉN, A., 2000, Experimentation in Software Engineering: an Introduction, Kluver Academic Publishers, Norwell, Massachusets.

[31] ZAFFER, A., SHAFFER, C., EHRICH, R., et al., 2001, NetEdit: A Collaborative Editor. Report TR-01-13, Computer Science, Virginia Tech, United States.

Exploring Coordination Structures in Open Source Software Development

Chintan Amrit, J.H. Hegeman and Jos Van Hillegersberg
University of Twente, The Netherlands
{c.amrit, j.h.hegeman, j.vanHillegersberg}@utwente.nl,

Abstract

Coordination is difficult to achieve in a large globally distributed project setting. The problem is multiplied in open source software development projects, where most of the traditional means of coordination such as plans, system-level designs, schedules and defined process are not used. In order to maintain proper coordination in open source projects one needs to monitor the progress of the FLOSS project continuously.

We propose a mechanism of display of Socio-Technical project structures that can locate the coordination problems in open source software development. Using the tool TESNA (TEchnical and Social Network Analysis) that we have developed; we cluster the software and produce a display of the different software clusters as well as the people working on its constituting software classes. We then demonstrate the technique on a sample FLOSS project that is on the brink of becoming inactive.

1. Introduction

Distributed self-organizing teams develop most Free/Libre Open Source Software (FLOSS). Developers from all over the world rarely meet face to face and coordinate their activity primarily by means of computer-mediated communications, like e-mail and bulletin boards [1, 2] . Such developers, users along with the associated user turned developers of the software form a *community of practice* [3]. The success or failure of open source software depends largely on the health of the health of such open source communities [4, 5]. Most of the literature on open source software development attests its success and only a relatively small but growing number of empirical studies exist that explain how these communities actually produce software [2, 6, 7]. Also, not everything is ok with open source development projects. Out of 153579 projects registered in source forge, only 12.24% of the projects had attained stable status (has a stable version of their software) when we last checked in July 2007. For an IT professional or FLOSS project leader it seems to be crucial to know the status of the open source project in order to contribute or recommend the project [4]. To this extent we provide a set of STSCs which can be checked in order to see the coordination inconsistencies of the work being done in an FLOSS project.

deSouza et al recognize socio-technical patterns of work assignment among the open source community members [7]. In this paper, we extend this research further by identifying socio-technical coordination problems, what we call Socio-Technical Structure Clashes or STSCs [8] based on some of these patterns. We provide a theoretical framework along with a technique for the identification of such STSCs. We then show the occurrence of the FLOSS STSCs in one open source project called JAIM, in order to demonstrate the technique.

2. OSS Community Structure

Although there is no strict hierarchy in open source communities, the structure of the communities is not completely flat. There does exist an implicit role based social structure, where certain members of the community take up assume certain roles by themselves based on their interest in the project [3].

According to Crowston and Howison a healthy open source community has the structure as shown in Figure 1 with distinct roles for developers, leaders and users.

Core Developers: Core developers are responsible for guiding and coordinating the development of a FLOSS project. These developers are generally involved with the project for a relatively long period, and make significant contributions to the development and evolution of the FLOSS system.

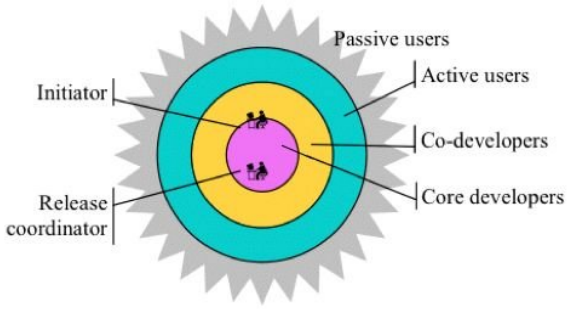


Figure 1: A healthy open source community (taken from [4]).

In those FLOSS projects that have evolved into their second generation there exists a council of core members that take the responsibility of guiding development. Such a council replaces the single core developer in second-generation projects like Linux, Mozilla, Apache group etc.

Project Leaders: The Project Leader is generally the person responsible for starting the FLOSS project. This is the person responsible for the vision and overall direction of the project.

Co-developers: Also known as peripheral developers, occasionally contribute new features and functionality to the system. Frequently, the core developers review their code before inclusion in the code base. By displaying interest and capability, the peripheral developers can move to the core.

Active Users: Active users contribute by testing new releases, posting bug reports, writing documentation and by answering the questions of passive users.

Each FLOSS community has a unique structure depending on the nature of the system and its member population. The structure of the system differs on the percentage of each role in the community. In general, most members are passive users, and most systems are developed by a small number of developers [2].

3. STSCs in FLOSS projects

A Socio-Technical Structure Clash (STSC) occurs if and when a Socio-Technical Pattern exists that indicates that the social network of the software development team does not match the technical dependencies within the software architecture under development. STSCs are indicative of coordination problems in a software development organization. Some of these problems (or STSCs) concerning development activities have been collected and described by Coplien et al. including a set of what they call Process Patterns to deal with these coordination problems. As the term process patterns is also used in business process management and workflow, we prefer to use the term Socio-Technical Patterns to refer to those patterns involving problems related to both the social and technical aspects of the software process [8]. de Souza et al. identify the following 2 socio-technical patterns by mining software repositories [7]:

- Core Periphery Shifts: In a healthy FLOSS project the peripheral developers more from the periphery of the project to the core, as their interest and contribution in the project increases [4, 7, 9]
- Code Ownership: In a healthy FLOSS project, the ownership of important code changes from a group of developers to one single developer [7, 10]

Therefore, the STSCs based on these Socio-Technical Patterns would be:

1. If the developers in the periphery do not move towards the center core
2. If the developers in the core move to the periphery of the project
3. If the ownership of important modules are shared by a large group of developers with no developer taking sole responsibility

From now on, we would refer to these STSCs as FLOSS STSC 1, 2 and 3 respectively.

4. Detection of STSCs in FLOSS

In order to detect STSCs related to FLOSS projects we used a clustering algorithm based on the algorithm by Fernandez [11] and later on used by MacCormack et al.[12]. We implemented this algorithm (see Appendix) to cluster the software components into 10 clusters, in such a way that the clustered cost given by:

$$CC(i) = \sum_{j=1}^{n} (SDM(i,j) + SDM(j,i)) * size(i,j)^2$$

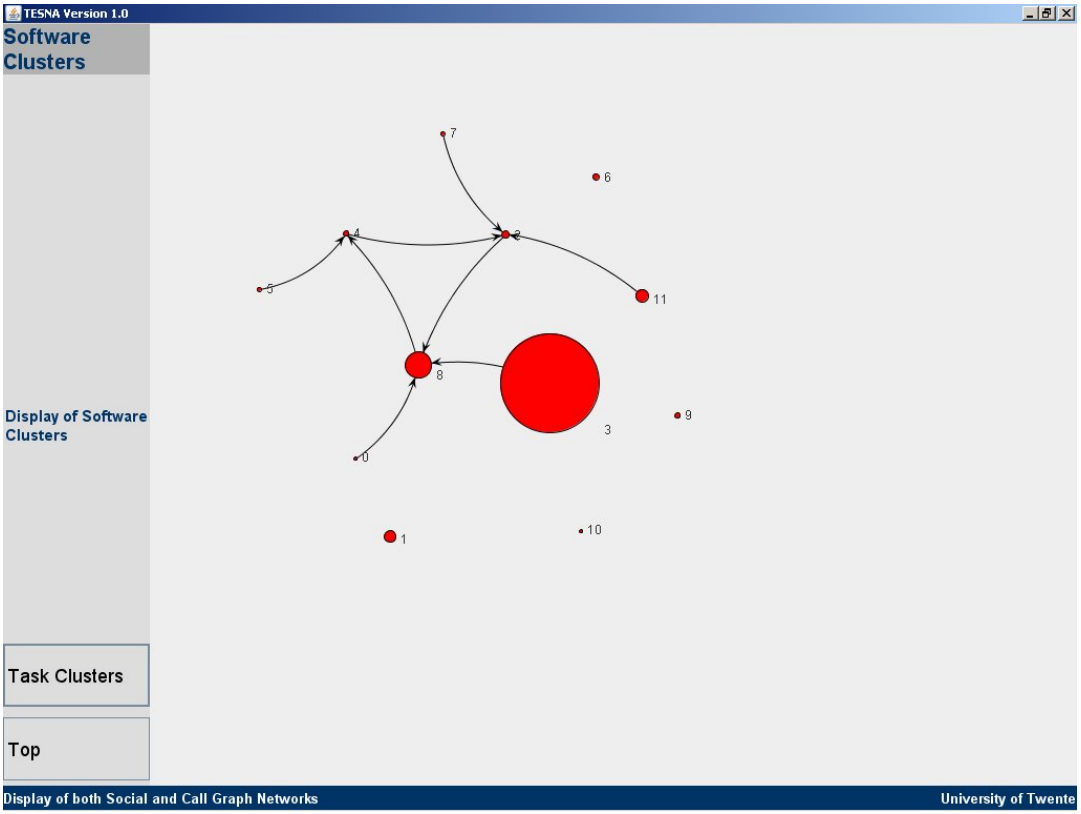


Figure 2: Class clustering of the JAIM software

was less than one. The resulting software clusters were as shown in Fig 2. We then included the author information of the components (extracted from the project's software repository (SVN)) in the same diagram and displayed the authors of the individual code modules as authors of the respected clusters (in which the code modules lay), as seen in Fig 4. As this clustering method is based on the dependencies between, the software components, the central cluster would represent the most dependent components of the software, or in other words the software core. Thus, the structure of the clustered software graph would represent the actual core and periphery of the software architecture. It has to be noted that this break up of core and periphery is based on software dependencies and could be different from that which was designed (especially in commercial software development, which generally follows traditional process of designing, developing and testing cycles). In this paper we trace the co-evolution of the project and the communities [9] and show the method of detecting FLOSS related STSCs by looking at the author-cluster figure (Fig 4-7) at equal intervals in the development lifetime of the project.

5. Empirical Data

In order to show the occurrences of the three STSCs we searched Sourceforge and selected a project called JAIM. The reason behind selecting JAIM was that it has a low activity percentage for the last year, and has no write activity in the repository (SVN) after 2006 (Fig 3). The public forums had 4 messages of which two were welcome messages. The mailing lists were empty. So were the Bug reports, support requests, patches and feature requests. This lack of activity showed that the project was heading towards becoming inactive. Therefore, we wanted to investigate the task structures over the period of the project, and see if we can find the three STSCs in this project.

TESNA can only display the dependencies between java modules, hence any of the .html documentation or java script files cannot be seen in the class cluster displays. Therefore, developers working on any of the non-java files will appear to be unconnected in the author-cluster displays. The software releases did not change the dependencies between the components in the releases 3 through 10, so the call graph and consequently the cluster graph of the software remained the same. So we just had to add the changed

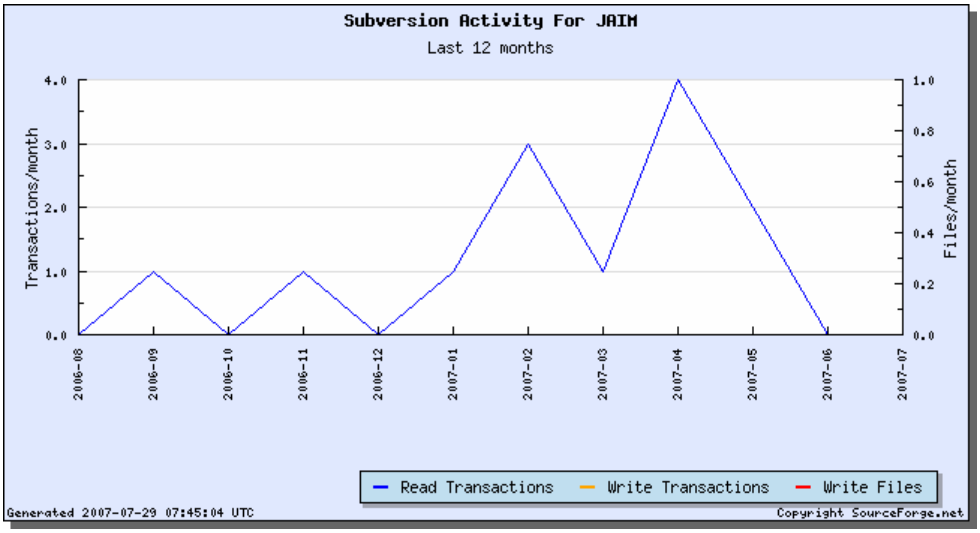


Figure 3: Subversion activity of JAIM (taken from Sourceforge)

author information in the TESNA tool in order to get the picture of which author was changing which module, belonging to the particular cluster.

6. Discussion

Using the tool TESNA[1] that we have developed we generated the author-cluster diagrams. We analysed the various write revisions in the repository (SVN) in order to detect STSCs mentioned in section 3. The latest revision of the project was 10, so we analysed revisions at equal intervals namely revisions 3, 5, 7 and 10. We have used TESNA to analyse the author-cluster diagrams qualitatively. We think approach is best suited in detecting such STSCs.

The first thing we notice in the author-cluster diagrams is that only 3 developers contributed to the development in the revisions 3, 5, 7 and 10 we observed namely, *coolestdesignz*, *root* and *dingercat* though 6 developers were listed as the developers in the project had 6 developers listed. Of these developers *coolestdesignz* is listed as the project administrator.

In revision 3 (Fig 4), we notice the developer *coolestdesignz* altering files in the core cluster 3 and the peripheral cluster 6. The developer *root* is also visible but is not visible changing any file in any cluster in the diagram as he/she would have changed/contributed to a non-java file. Since the file(s) is not in java, it's an .html or java script file which would mean that root is a peripheral or co-developer for this revision.

From figure 4 to 7, we see the occurrence of FLOSS STSC 1 (section 2) as the peripheral developer *root* does not move to the core.

We notice FLOSS STSC 2 as core cluster developers *coolestdesignz* and *dingercat* move to the peripheral region, as *coolestdesignz* disappears after revision 3 (Fig 4), while *dingercat* moves from being a core developer to being a co-developer in revision 10 (Fig 7). FLOSS STSC 3 can be seen between revisions 3 and 5 (Fig 4 and 5) where *coolestdesignz* as well as *dingercat* modify the core cluster with no single developer responsible for the modules in the core cluster.

[1] http://tesnatool.googlepages.com

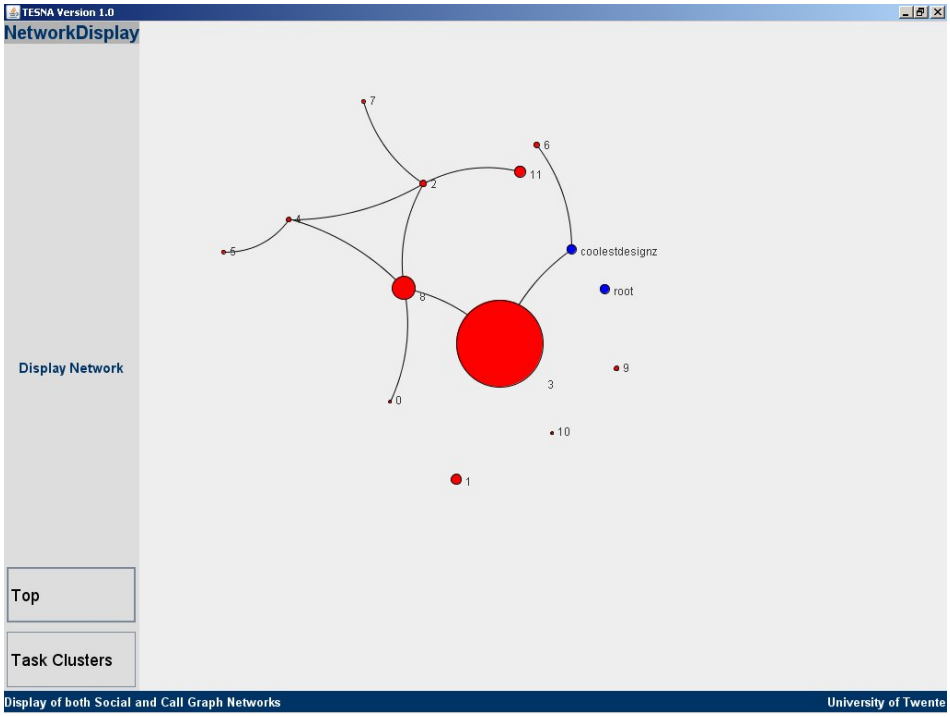


Figure 4: Class Clusters along with Author information for revision 3

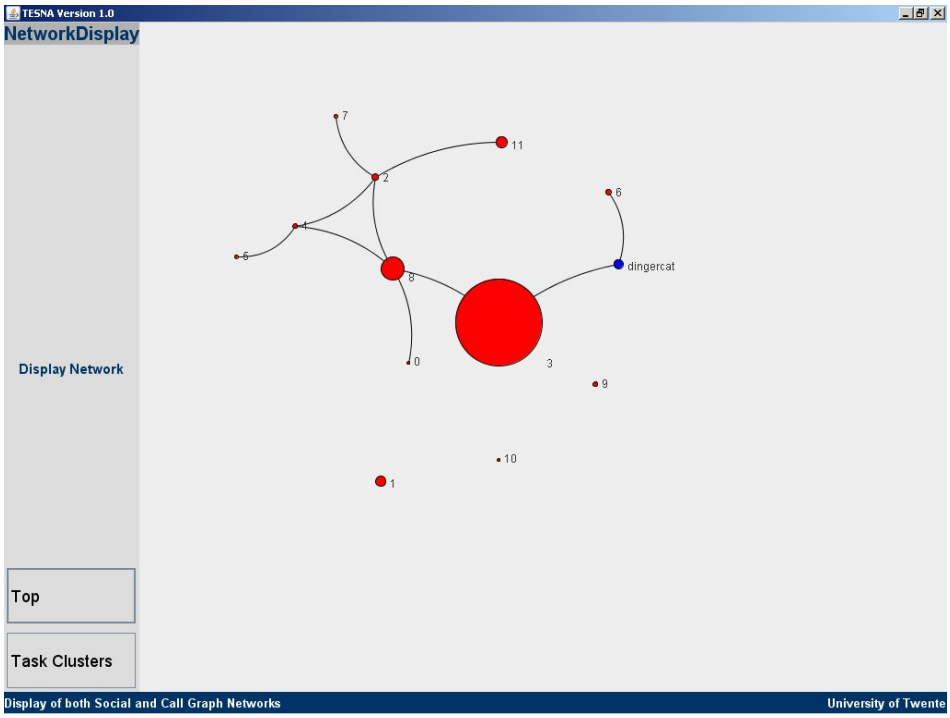


Figure 5: Class Clusters along with Author information for revision 5

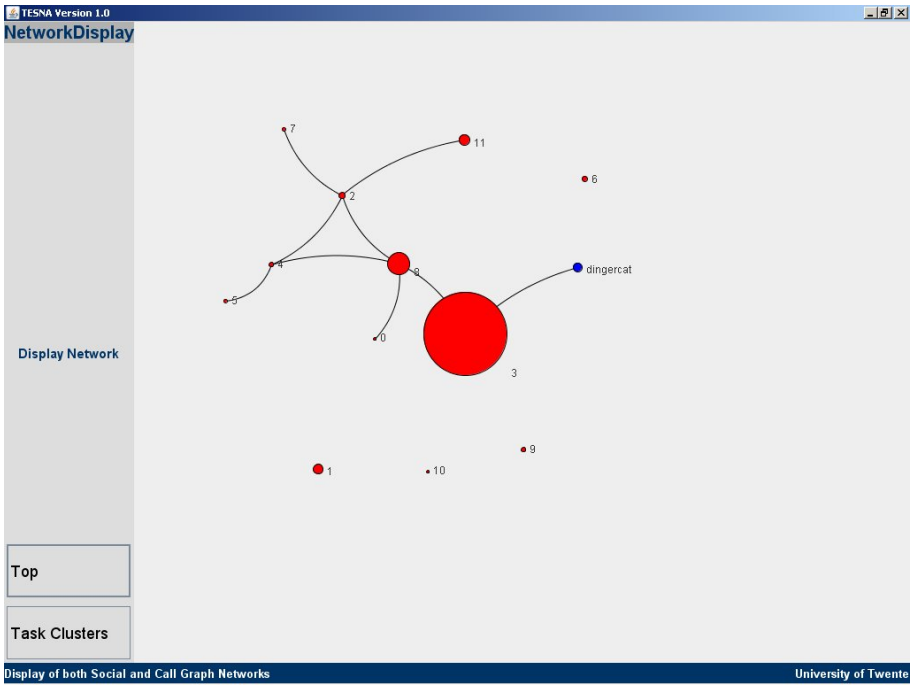


Figure 6: Class Clusters along with Author information for revision 7

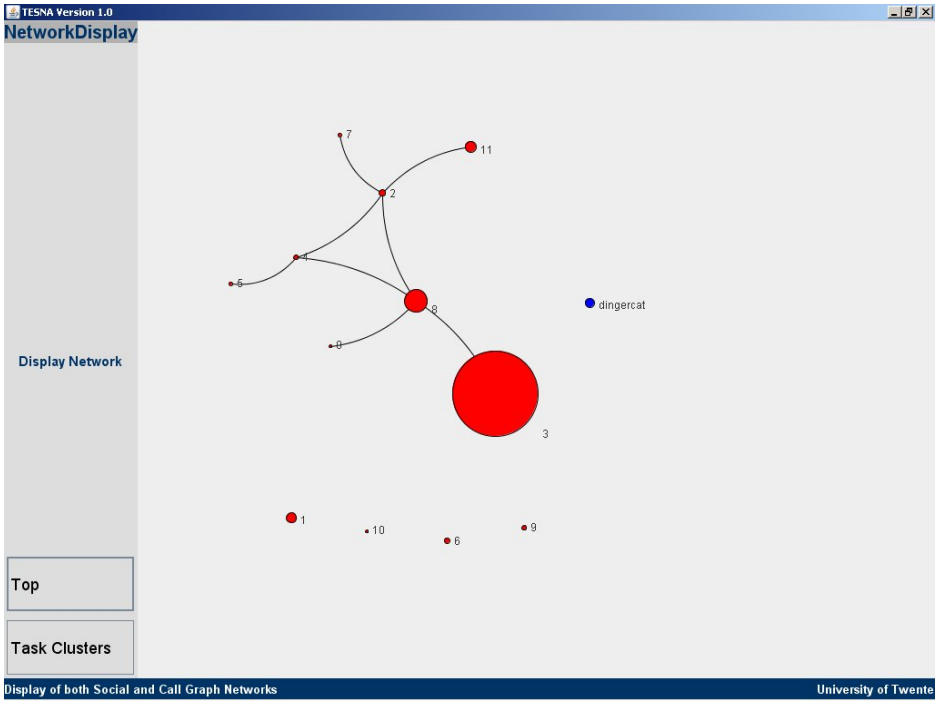


Figure 7: Class Clusters along with Author information for revision 10

7. Conclusion

In this paper, we have tried to come up with STSCs based on FLOSS literature. We have showed a

technique (a clustering based display mechanism) that can be used to detect STSCs in FLOSS projects. We have tried to demonstrate this technique by detecting STSCs in an open source project JAIM. The project JAIM is in the beta stage of development and has all the signs of joining the ranks of an inactive project in the Sourceforge database. Through the detection of STSCs, we plan to inform the project leader (of JAIM for example) as well as potential interested developers on the health of the open source project. Though, there are other ways of detecting the health of an open source community [4, 5], the techniques they use do not display the actual task allocation of the FLOSS project.

In this paper, we display a technique using the actual mining of data from the code repositories to see the actual contributions of the developers. The greater the number of STSCs found in the project the worse would be its health. Future work could involve the identification and detection of many more STSCs in different open source projects enabling project managers to manage the FLOSS development process in a better way.

References

1. Raymond, E., *The Cathedral and the Bazaar*. Knowledge, Technology, and Policy, 1999. **12**(3): p. 23-49.
2. Mockus, A., R.O.Y. T Fielding, and J. D Herbsleb, *Two Case Studies of Open Source Software Development: Apache and Mozilla*. ACM Transactions on Software Engineering and Methodology, 2002. **11**(3): p. 309-346.
3. Ye, Y. and K. Kishida, *Toward an Understanding of the Motivation of Open Source Software Developers*. 2003. p. 419-429.
4. Crowston, K. and J. Howison, *Assessing the Health of Open Source Communities*. 2006, IEEE Computer Society Press Los Alamitos, CA, USA. p. 89-91.
5. Crowston, K., H. Annabi, and J. Howison, *Defining Open Source Software Project Success*, in *International Conference on Information Systems*. 2003.
6. Yamauchi, Y., et al., *Collaboration with Lean Media: how open-source software succeeds*. 2000, ACM Press New York, NY, USA. p. 329-338.
7. de Souza, C., R. B., J. Froehlich, and P. Dourish, *Seeking the source: software source code as a social and technical artifact*, in *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*. 2005: New York, NY, USA. p. 197--206.
8. Amrit, C. and J. Hillegersberg, *Mapping Social Network to Software Architecture to Detect Structure Clashes in Agile Software Development*, in *15th European Conference of Information Systems*. 2007.
9. Ye, Y. and K. Kishida. *Toward an understanding of the motivation Open Source Software developers*. in *25th International Conference on Software Engineering*. 2003: IEEE Computer Society Washington, DC, USA.
10. Coplien, J., O. and N. Harrison, B. , *Organizational Patterns of Agile Software Development*. 2004: Upper Saddle River, NJ, USA.
11. Fernandez, C.I.G., *Integration Analysis of Product Architecture to Support Effective Team Co-location*. 1998, MIT: Boston, MS.
12. MacCormack, A., J. Rusnak, and C.Y. Baldwin, *Exploring the structure of complex software designs: An empirical study of open source and proprietary code.* Management Science, 2006. **52**(7): p. 1015-1030.

Appendix

Clustering Algorithm:

```
clusters findClusters (param inputDSM) {
   Set clusters = one cluster for each module;
   int cost = calculate initial total cost;
   int failedattempts = 0;
   Do {
      m = a random module from inputDSM;
      For every element c from clusters {
         calculate bid of c for m;
      }
      cmax = highest bidding cluster;
      best_bid = bid from cmax;
      If (best_bid > 0) {
         Reassign m to cmax;
         For every element c from clusters {
               If (c is empty) remove c;
         }
         cost = cost – best_bid;
         failedattempts = 0;
      } else {
         failedattempts++;
      }
   } while (failedattempts <= FAILED_LIMIT)
   Return clusters;
}
```

Tool Support for Distributed Software Development

The past – present – and future of gaps between user requirements and tool functionalities

Miles Herrera* and Jos van Hillegersberg**
* *Requirements and Process Engineer, Atos Origin The Netherlands, m.herrera@atosorigin.com*
** *Professor of Information Systems, University of Twente, The Netherlands, j.vanhillegersberg@utwente.nl*

Abstract

This paper presents the past, present, and our view on future user requirements and tool functionalities supporting Globally Distributed Software Teams and highlights the changing emphasis in these user requirements..

1. Introduction

Ten years ago Gartner Group predicted that by the year 2003 [1] about 140 million global business users worldwide would be involved in remote work. We all have been part of the fulfillment of this 'prophecy' and it is likely that this figure will continue its exponential growth. The software sector is increasingly adopting globally distributed work as a means to utilize skilled resources around the world, speed up development and save costs. However, Globally Distributed Software Teams (GDST) face new challenges that were not present in the days of co-located development. Most of the tools created to support the work of software teams have not specifically been designed for GDST. Also, only recently research is giving more insight into effective practices and processes for GDST. As a result, establishing a comprehensive tool environment for GDST is a challenge. As the market is rapidly evolving, this paper provides a framework for assessing GDST requirements. We use this framework to provide an overview of the past, present and future of GDST tools. The results reveal that the requirements are shifting from support for collaborative coding and requirements management to communication, process management and collaboration support. In addition there is an increasing need for various types of integration: between both generic and special purpose tools, between planning and monitoring tools, and between technical (code and design) and social (communication and management) tools. The paper first introduces the framework and then uses the framework to analyse the past, present and future GDST requirements and tool offerings. We analyze the past requirements and tool market using our earlier survey of the tool market conducted from 1997-2002. We base our analysis of the present and future on recent developments in requirements and the tool market (2002-2007) and our view on future.

2. Scope/ methodology

In this paper we present the changes of focus in the past 10 years when looking at GDSD projects. By taking three different views in the analysis – management, tool vendor, and research – we present different patterns for further research. For the analysis the model of Globally Distributed Software Engineering Environment (Kotlarsky et al., 2001) is used as a framework. This framework is used to highlight changing areas of attention over the past ten years.

The analysis is based on a survey of tool functionalities, empirical study (including experience in the field, observation, and semi-structured interviews) and literature study.

3. A framework for tool requirements

Kotlarsky et al., (2001) integrate several earlier studies into a generic model for requirements for Globally Distributed Software Engineering Environments. We will use this framework for our analysis and summarize it here. It includes the following categories of user requirements (UR):

UR1: Describe and maintain data information during the product development life cycle.

UR2: Coordinate and control during a project and across projects.

UR3: Have common/remote accessibility to the project environment.

UR4: Negotiate and reach consensus with other GDSD team members.

The inside area of the model in Figure 1 illustrates the common workspace, and contains the product, process, and project organization structures that are

interconnected by plans. They are used to describe and maintain data information required by (remote) project members during the product development life cycle (UR1).

The *coordination* and *control framework* (UR2), surrounding the *common workspace*, includes the following activities:

- Planning, scheduling, allocation
- Monitoring
- Outsourcing management
- Configuration management
- Tracing management
- Constraint management
- Progress measurement and ensuring quality
- Risk management
- Version management

All these activities apply to relevant components of the *common workspace*. Consequently, most of the coordination and control activities could be considered as *product*-, *process*- and *organization*-oriented. (e.g. Tracing management provides mechanisms to trace user requirement – to make sure that they are implemented in design and later in coding).

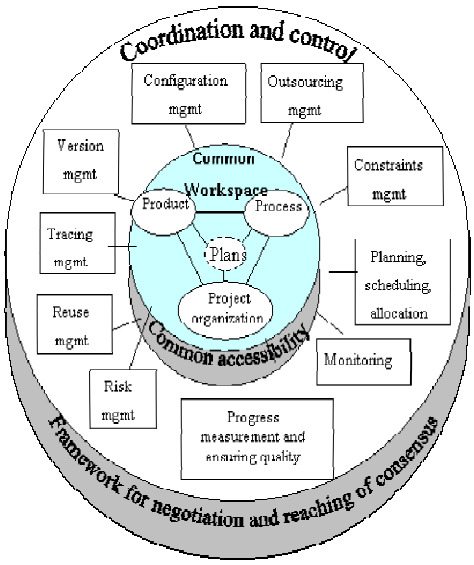


Figure 1_The Model of GDSD Engineering Environment

The third dimension of the model, the dark gray areas, displays the requirements of the *common workspace* and *coordination and control framework* necessary to make it available for working in a distributed setting and support remote communications:

Common accessibility (UR3) is required in a distributed environment to make *product*, *process*, *project organization* structures and *plans* transparent (visible) through remote sites and to allow remote access to the required data/information.

Framework for negotiation and reaching of consensus (UR4) provides functionality needed to support remote communications. It applies to all elements of the coordination and control framework and common workspace as well. On the technical level, the above-mentioned functionality will be provided by collaborative technology.

Together, *common accessibility* and *framework for negotiation and reaching of consensus* are supposed to eliminate (or at least reduce) the perception of distance, and unite remote project members creating a joint project environment.

4. Past – Present – Future

This section presents our analysis of the different periods. Each paragraph explains the characteristics from three different points of view: management, tool vendor, and research.

4.1. The past (1997 – 2002) – Product oriented vendor driven market

Management

Around the Millennium, driven by the fast adoption of Internet, the working environment of software development projects changed. Projects became more globally dispersed, run over different time zones, and across organizations [2]. Also, as teams became increasingly dispersed, new challenges were introduced like the standardizing and sharing designs, sharing ideas and experiences, and sharing/storing of work in a shared repositories. To meet these challenges GDSD teams highly relied on optimal tool support.

Traditionally software development teams have used Computer-Aided Software Engineering (CASE) tools and Version Management Systems (VMS). However, these tools were usually not designed to support dispersed teams and issues were resolved in face-to-face meetings.

During this period much attention was paid on tools and less on the underlying process. As there was a lack of standardization of tools and limited alignment with the development process, projects were often difficult to manage.

Tool vendor

Our earlier research of the tool market conducted in 2002 [3] was aimed at analyzing the gap between GDST user requirements and tool offerings. We investigated how a GDST tool environment consisting of a combination of tool functionalities of CASE tools, GSS tools, and VMS tools could meet the requirements. Ideally the resulting user requirements of each activity identified in the model of GDSD (Fig. 1) should be supported. For the research three commercial CASE tools were chosen, Together ControlCenter version 5.5 (Borland), Rose version 2001 (IBM Rational), and TauUML (Telelogic). A fourth tool (non-commercial), JComposer, was added to the list and represented the category of research tools. The VMS tools were SourceCast (Collabnet), ClearCase (IBM Rational), PVCS Version Manager (Merant), and Visual SourceSafe (Microsoft). In order to extend the change tracking functionality, ClearCase was combined with ClearQuest and PVCS Version Manager with Tracker. The tools representing the GSS tools were Lotus Notes integrated with Sametime, MS Exchange 2000 integrated with Conferencing Server, SiteScape Forum, and eRoom 5.0 integrated with eRoom Real Time Services.

Table 1_ Enhancements to CASE tools

Feature	Secondary Empirical Research (tool evaluation) Enhancement	Primary Empirical Researc (case study) Enhancement
Sharing models, checkout file locking, visual merge, and difference reporting	Addition	
Comparison and merging of two versions		Addition
Sending/receiving email	Integration	
Chat and Whiteboard	Integration/Addition	Integration/Addition
Documentation generation	Addition	
Audits and metrics	Addition	
Change Management System	Integration	Integration
Integration between source code, VMS, Change Management tool, and email		Integration

Our study concluded that no single tool came close to covering the needs of GST. Also, no combination of tools could provide an ideal environment. Tools in various categories had overlapping functionalities and also gaps existed. Moreover, as a result of limited standardization integration between tool functionalies was lacking or primitive. The research concluded with advice on how to improve the existing CASE tools to fit better to the user requirements. This was done by closing the gap in functionality by adding functionality to the tool or integrating with other tools (Table 1).

Research

Mostly outside the area of software development Group Support Systems (GSS) emerged specially focused on supporting teamwork. However, the integration or support of GSS functionality in CASE tools was in this period in its infancy and received more attention from the research community then from the vendor community. Examples are the Knight tool[4] and JComposer[5].

4.2. The present (2002 – 2007) – Process oriented vendor driven market Quality – Costs – Time

Management

The last five years and after the ‘ Internet Bubble’ have been characterized by the emphasis on the *development process* rather then tooling. Having the process as the central nerve in GDSD, management is able to select the right tools. Companies fighting to maintain their competitive advantage have become more rigorous in keeping the balance between quality – costs – and time to market. The ‘hype’ behind outsourcing and off shoring in search of lowering development and services costs, have forced IT management to improve and redesign their software development process.

Project risk management has also received more attention in the Coordination and Control area of the framework. Risk management introduces three ‘new’ risk factors in GDSD – trust, culture, and collaborative communication. Cultural differences are primarily a risk source (an origin for problems), whereas trust is primarily a risk driver (a manifestation for problems). Communication can be a source (e.g. mistranslation of requirements) or a driver (manifesting lack of management support), or both[6]. To implement risk management successfully the theoretical aspects of project risk management have to be integrated with practical challenges of the organization. At the end risk management process will succeed by changing the organizational culture to motivate the individual. Cultural changes require time and repetition before they are firmly embedded in the organization [7].

Tool vendors

A market consolidation caused by the take over of several small players by the market leaders IBM and Microsoft has reshuffled the tool vendor landscape. The two important market players have been working on an integrated collaboration platform, respectively IBM Workplace with Eclipse Lotus Notes/Domino technology and Microsoft Vista [8]. Community software allows the usage of low cost single tools for collaboration between project sites. Tools like jabber &

msn chat, groupware like yahoo groups, Google shared docs en spreadsheets, the new Sharepoint/Office 2007, to mention a view. Experiences from nine different GDSD projects at Siemens show promising results about the usage of synchronous tools like chat and asynchronous tools like wikis and discussion boards when working in different time zones [9]. Another study [10] reaffirms the effective usage of synchronous communication. Content analysis shows that chat was used for work discussions or for articulation work to coordinate projects and meetings, and to negotiate availability. This resulting in a better fit to the common workspace requirements (UC4) in current GDSD.

Both tool vendors also compete in the tools supporting the software development process.

IBM has gained market share by supporting the Open Source and Java community and becoming the company behind Eclipse platform and integrating different new tools, existent tools and acquired Rational tools. Microsoft is coming up with a new integrated suite on top of Visual Studio, called Visual Studio Team Systems and supporting dotNet.

The change in tool vendor strategy also reflects the move from a tool centric strategy to a process centric strategy.

Microsoft has build its new Visual Studio Team Systems around its own process, while IBM has been developing different flavors of the Rational Unified Process (RUP). RUP has been on the market for more than a decade and has been remarketed in the Method Composer (commercial tool) and the Eclipse Process Framework[1] (EPF) (open source tool). Interesting in EPF is the flexibility to build company specific processes and integrate this into different software tools.

Research

GDSD tools have complex issues to address, such as user interface design, varying levels of collaboration requirements, varying expectations between developers within a group, support of multiple artifact types, and potentially multiple views of artifacts. There are also technical aspects to address such as concurrency control and distributed system design, along with the standard software engineering technicalities such as parsing, semantic modeling and source code management [11].

On the other hand, the ideal tool to support collaborative software engineering is still a challenge for tool vendors and researchers. Research projects like CAISE architecture provides an infrastructure with the potential to support the entire software engineering process [11]. In opposite to commercial tools, the CAISE architecture is not built on top of a source code repository. This brings new light to the integration discussion, but at the same time researchers recognize that the costs involved for the development of a commercial platform are high and without a clear business case for tool vendors.

4.3. The future (2007 -) – Orchestrating global development

Unfortunately nobody can look in the future. In this paragraph combine some studies that have attempted to do so, including several reports by Gartner on future developments around GDSD.

Management

Gartner predicts that through 2009, 30 percent of Global 2000 enterprises will have reached a high level of maturity (institutionalized) in supporting telework practices, services, and infrastructures[12]. This trend will help managers to save on different type of employee costs (e.g. Cisco Systems' distributed work strategy reports a return on investment of 300 percent resulting from real estate savings and reduced employee turnover). Characteristics of this type of the institutionalized maturity level are: strategic business imperative, 100 percent participation, fully integrated workplace organization, and virtual work as a 'way of life'. During next years management will be focused on managing the different organizational structures of GDSD projects, new development processes and the way projects are managed [13].

Collaboration between teams will become more mature and one can assume that GDSD teams will benefit from it.

In addition other challenges like increasing complexity in software and architectures, technological changes (paradigm shifts), knowledge management, growing cultural and language differences (e.g. upcoming outsourcing countries like China) and IT workforce management [15] will affect the further development of tool support for GDSD projects.

Tool vendor

In the future, shared and customized workspaces will give more value if integrated in the processes and needs of the companies [14]. The needs are different by teams and by companies and no one tool vendor will provide all functionalities. For example, peer-to-peer products such as Microsoft Office Groove 2007 support offline collaborative work among dispersed

[1] www.eclipse.org

teams, while SiteScape's offering includes native workflow functions.

Research

Recent developments in research will also have an impact on tool environments of the future. After decades of emphasis on design and coding tools, visualization of architectures, business rules and requirements receives increasing attention. Under the umbrella of Model Driven Architecture (MDA), more attention is being payed to models that can be semi-automatically translated into other abstractions, providing a different view on the system.

Highly specialized team members which can be located anywhere on the globe should be enabled to contribute to a project on highly flexible contracts, e.g. conduct a security audit of a system design. To empower such work, development environments should provide these individuals with quick insight into the proper system views and at the same time only provide visibility to parts of the system needed to conduct the task.

The increasing complexity of systems requires tools that provides powerful monitoring and analysis capabilities. E.g. tools that provide automatic testing, longitudinal analysis of system complexity metrics etc.

Although tools that support collaboration have been around for decades, these have usually not been integrated to technical CASE. Recently, we witness research into tools that combine ideas such as social network analysis, to architecture and design tools.

To manage the vast amount of information present and created during a systems development project, ideas from knowledge management and information retrieval will be integrated into future environments, e.g. to retrieve reusable requirements or design specifications.

Finally, standardization or advanced translators need to enable integration between various advanced tool components. Each organization will have a wish to integrate various types of generic and special purpose tools, planning and monitoring tools, technical (code and design) and social (communication and management) tools.

11. Conclusion

Requirements are shifting from support for collaborative coding, and requirements management to communication, process management and collaboration support (four user requirements introduced in paragraph3) . In addition there is an increasing need for various types of integration: between both generic and special purpose tools, between planning and monitoring tools, and between technical (code and design) and social (communication and management) tools. As common software architecture for one generic tool is very expensive and different tool vendors like Microsoft and IBM are providing collaboration tools on the desktop level, it is assumed that each market player will focus on specialization rather than generalization and will in the long run support the integration requirements.

With the further globalization new management organizational structures for GDSD teams emerge and become interesting topics for further research. In these new structures tool support and mainly collaborative support remains important, software development process will become more mature as the need for governance increases and common understanding of requirements between different locations, cultures, and knowledge levels.

We also see the service oriented architectures (SOA) and service on demand as an interesting topic of research that will ad a new dimension to the model of GDSD and the way product, process, and organization are approached.

10. References

[1] Carmel, E. *'Global Software Teams: Collaborating Across Borders and Time Zones'*. Upper Saddle River, NJ, Prentice-Hall PTR. 1999

[2] Fenema. *'Exploring the Nature of Globally Distributed Collaboration'*. 2001

[3] Herrera Miles. "*Globally Distributed Software Development Project: Gap between CASE tools and GSS tools*". Master Thesis Erasmus University. May 2002.

[4] Damm, Ch.H., Hansen, K.M., and Thomson, M. *'Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard'*, in: Chi Letters, Vol. 2, issue 1, pp 518-525. 2001

[5] Grundy, J. *'Engineering Component-based, User configurable Collaborative Editing Systems'*, Thesis Publishers Amsterdam, pp 16. 1998

[6] Mohtashami, M.; Marlowe, M.; Kirova, V.; Deek, F.: "Risk Management for Collaborative Software Development". Information Systems Management. Fall 2006.

[7] Kwak, Y.H.; Stoddard, J.: *"Project risk management: lessons learned from software development environment."* . Technovation 24 p.p. 915 – 920, 2004.

[8] Mann, J. and T. Austin. Management Update: *Microsoft and IBM share Similar Collaboration Goals, but Follow Different Paths*. Gartner Research ID Number G00132087 2005.

[9] Herbsleb, J.; Paulish, D.; Bass, M.: *"Global Software Development at Siemens: Experience from Nine Projects"*. ICSE'05. May 15-21, 2005.

[10] Handel, M. and J. Herbsleb.; *"What Is Chat Doing in the Workplace"* . CSCW'02. 2002

[11] Cook Carl, Churcher Neville: *" Constructing Real-Time Collaborative Software Engineering Tools Using CAISE, an Architecture for Supporting Tool Development*". 2006.

[12] Bell, Michael: *Gartner's Telework Maturity Model Defines the Stages Toward Telework Effectiveness:* 2006: Gartner Research ID Number G00136640.

[13] Herbsleb, J.: *"Global Software Engineering: The Future of Socio-technical Coordination"*. FOSE'07. 2007.

[14] Smith, David Mario: *"Customize Shared Workspace Software to Users' Needs"*. 2006: Gartner Research ID Number G00141049.

[15] Morello, Diana: *"IT Workforce Management: Prepare for a Future Unlike the Past"* .Gartner Research ID Number G00126450 2005.

International Workshop on Tool Support and Requirements Management in Distributed Projects (REMIDI'07)

Eva Geisberger, Patrick Keil, Marco Kuhrmann
Technische Universität München, Institut für Informatik, I-4
Boltzmannstrasse 3, 85748 Garching b. München, Germany
{geisberg,keil,kuhrmann}@in.tum.de

Abstract

Today, distributed projects, often subsumed under terms like global software development (GSD), global collaboration, offshoring etc. are common ways to overcome time and budget restrictions or lack of personnel. Thus, today's projects take place in a global context. But developing software with geographically distributed teams presents a unique set of challenges that influence virtually all aspects of a project and make them more complex. This workshop addressed topics relevant in multi-site projects like tooling, process support, economic aspects, project management and collaboration and communication.

1. Introduction

Developing software with geographically distributed teams presents a unique set of challenges influencing all aspects of a project and increasing complexity. In these projects, many aspects of project members' daily work have to be reconsidered. For example, there is a lack of proven requirements engineering (RE) concepts and practices in the context of global software development (GSD). Also aspects like knowledge management and project tracking ask for appropriate tools to help project members reaching their goals.

Besides other challenges, planning, coordinating and controlling of requirements engineering, implementation and testing in distributed settings are far more complex than in one-site projects.

First, the processes of requirements elicitation, system modeling, coding, testing and rollout need to be planned and organized differently. Second, the methods used to share and discuss early design ideas, coding decisions or test results need to take into account the fact that some project members involved in these phases and tasks are spread over multiple sites and organizations and don't have contact to end-users. For all these tasks, a sophisticated tool chain is needed.

Experience shows that an adequate tool chain increases efficiency and success of distributed projects and need to be properly supported. This is why we focused on this aspect. This workshop walked through methods, tools and concepts that are or should be used in requirements engineering, software development and testing in global software development projects.

2. Topics

One of the main objectives of this workshop, held at ICGSE 2007 in Munich, was to structure the major research topics and to define a research agenda for further work in the area of tool support in distributed system development. To this end, we solicited position papers presenting field reports, case studies, analytical frameworks and key research questions, which serve to improve our knowledge on the different aspects of infrastructure and tools in a GSD context, e.g.:

- *Tooling*: Which are the issues inherent with GSD? How to support global development project with tools in an appropriate way? Are the tools for project management or workflow-support different to those used in on-site projects?
- *Administration and tracking of architectural documents*: What are the consequences for the process and the design tools if the process of architecture definition is distributed?
- *Process support*: What does an adequate process for distributed development look like and how should it be supported by tools and techniques?
- *Economic aspects*: How can we evaluate the efficiency of geographically dispersed requirements engineering, also compared to on-site projects? What is Return on Investment in dedicated tools in distributed development?
- *Project management*: Which tools can help to plan, control and track a project? Are risk management or workflow management tools different to those used in on-site projects?
- *Collaboration and communication*: How do RE and software development need to be organized when teams are spread over two or more sites? How can projects achieve efficient collaboration? What are the lessons learned on tools and infrastructures for aligning in RE, development or test?

3. Workshop Presentations and Discussions

The topics mentioned above were discussed based on presentations by participants. Four position papers, from ten authors based in four countries, were accepted to be published in the workshop proceedings (available in hardcopy from CTIT and in PDF format at http://www4.in.tum.de/~kuhrmann/remidi07.shtml).
The papers covered a wide range of topics, including:

- Communication Tools in Globally Distributed Software Development Projects
- Groupware System for Distributed Collaborative Programming: Usability Issues and Lessons Learned
- Sensitivity Analysis Approach to Select IT-Tools for Global Development Projects
- Requirements Management Infrastructures in GSD

One of the main objectives of this workshop was to define structure the major research topics and to define a research agenda for further work in the area of tool support in Global Software Development. In addition to the papers from researchers and practitioners, we therefore invited two keynote speakers who have in-depth knowledge and manifold experiences with distributed development. Daniel J. Paulish of Siemens Corporate Research gave a thrilling introduction into the pitfalls and experiences of distributed software development. Rupert Stuffer, CEO of ACTANO Group reported on the challenges his company faces when developing software that is specifically used to help managers planning and controlling software development projects. Their presentations are also available at http://www4.in.tum.de/~kuhrmann/remidi07.shtml.

After the keynotes, paper presentations and a joint session with TOMAG 2007 workshop, the participants discussed the research areas which are most relevant for practitioners in their work in distributed software development.

A Sensitivity Analysis Approach to Select IT-Tools for Global Development Projects

Céline Laurent
BMW Group
Knorrstr. 148
80788 München, Germany
Celine.Laurent@bmw.de

Abstract

Globalisation is accorded increasing interest and importance in contemporary world affairs and industrial concerns. Subsequently, engineering projects involving different companies, cultures and disciplines are becoming more complex and require interaction and communication between all project participants. Collaboration-tools are instruments to support distributed projects successfully. However, selecting the correct IT-tools which correspond to the characteristics of globally-distributed projects is not an easy task. In fact, supporting global development projects in an appropriate way can be difficult, since it is influenced by many factors. For this reason this paper first proposes a taxonomy of influencing factors on global development projects and applies a sensitivity analysis method on empirical data to assess the degree to which influencing factors determine the choice of IT-tools.

1. Introduction

Globalisation implies a strong networked and distributed development process; it offers high potential for businesses due to its innovation, flexibility, cost reduction, time reduction, and improving quality. At the same time, it continues to be a challenge to utilise this potential nowadays. Practice has shown that various influences like spatial distance, differences in culture and language, inconsistency of processes, as well as intransparency of information and communication can be responsible for the lack of cooperation at the operative work level alone. Coordination and an exchange of information between participants in a distributed product development team are technically difficult and time consuming, especially when different locations and time zones further complicate communication.

Information and communication systems help such projects to become successful in supporting exchange between multiple partners. The most common way to support these standard processes is to use Groupware Systems or Computer Supported Cooperative Work (CSCW, cf. [1] and [2]). It's obvious, however, that every project has a specific character and that a single IT-tool cannot support each and every engineering activity. Moreover, choosing the most appropriate IT-support is quite difficult, since technology is continually in evolution and its complexity makes the depiction of precise requirements on collaboration software hard. This paper discusses the main influencing factors and their weight regarding the choice of IT-tools to support global development projects.

2. Method and instruments

To outline the method, we will first present the factors and the IT-tools under consideration. This will be followed by an empirical study which has been conducted to establish a correspondence between these factors and the IT-tools which have been selected. Finally, we explain the method used to determine which factors are the most influential.

2.1. Influencing factors

We base our analysis on a taxonomy of 16 influencing factors of distributed and interdisciplinary development projects according to [3]. Table 1 shows these factors with their parameter values. It combines the typical characteristics of distributed engineering

processes with their possible conditions according to [4][5][6], as well as the factors resulting from the interaction of different domains presented in [7].

Table 1. Influencing factors

N°	Factor	Possible Values		
1	Number of cooperation partners	Low	Average	High
2	Location	Same Location	Same company	Other country
3	Skill level in the agreed language	Insufficient	Satisfying	Excellent
4	Type of engineering process	Parallel	Sequential	
5	Organisation and company's culture	Same Business Unit	Same company	Other company
6	Size of the organisation	Small	Middle	Large
7	Intensity of collaboration	Low&irregular	Integrated	
8	Distribution model	Equal	Unequal	
9	Number of interfaces	Low	Average	High
10	Access to data	Very Difficult	Difficult	Easy
11	Skill level in the use of IT	Low	Average	High
12	Influence of time	Sufficient	Insufficient	
13	Methods and instruments	Same	Different	
14	Vocabulary	Same	Different	Contradictory
13	Methods and instruments	Same	Different	
15	Standards and laws	Same	Different	
16	Dependencies on own domain	Yes	No	

2.2. Empirical study

In the empirical study 18 standard IT-tools which support global development projects have been rated by experts for their support of cooperation projects with respect to the influencing factors of Table 1. The survey was carried out by interviewing experts in distributed projects, interdisciplinary projects, cooperation projects, and Information Technology (IT). The results of this empirical study are presented in [3].

A brief glance of the entire results (c.f. [8]) shows that some influencing factors have a larger impact on the selection of the experts than others. Moreover, the same factors do not always have the same impact on the choice of one tool or another. To evaluate the significance of each factor we have to quantify the dependence of the output on the input parameters. To this end a sensitivity analysis is carried out.

2.3. Sensitivity analysis

In general, sensitivity analysis is the study of how the uncertainty in the output of a model can be attributed to different sources of uncertainty in the model input. Various questions can be answered by means of a sensitivity analysis. According to [9] it is important to specify the purpose of the analysis before starting it. A sensitivity analysis can be carried out to simplify models and fix factors, to prioritise factors, to identify critical factors, etc. We want to determine a ranking of the influencing factors. In this context one setting of sensitivity analysis is interesting: factor prioritization.

In Factors Prioritisation (FP), the question addressed is: which factor, once fixed to its true albeit unknown value, would provide the greatest reduction in the uncertainty of the output? This factor is then the most important factor. Likewise the second most important factor can be defined and so on (cf. [10] and [11]). The ideal use of the setting FP is for the prioritisation of research, which is also one of the most common uses of sensitivity analysis in general. Given the hypothesis that all uncertain factors are susceptible to determination, setting FP allows the identification of the factor that most deserves a better experimental measurement in order to reduce the target output uncertainty the most. Hence, we can determine those factors that should be measured most precisely in our empirical study. (cf. [12]).

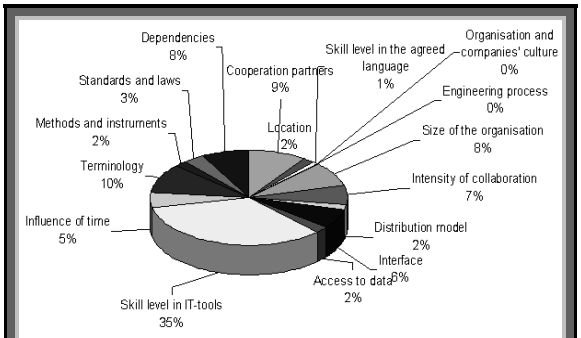
There are various methods of sensitivity analysis available. The Fourier amplitude sensitivity test (FAST) is a commonly-used approach that is based on the Fourier series which describes the output functions (cf. [12]). When the results of the FAST are applied to our influencing factors, they not only give a qualitative ranking, but also a quantification of these factors.

We use the sensitivity analysis tool SimLab (cf. [13]) for the analysis. According, we must define all the inputs parameters which are needed and their value sets. The tool then generates the samples needed for the FP analysis. Our goal is to determine with the help of the expert data which factors have more impact on the selection of each tool.

The input factors are shown in Table 1. Their value sets are derived from the results of the survey in [8]. Experts gave us a rating for each tool and each value of a factor. These ratings compose the value sets of the factors. With the help of any simple model which uses each factor once with the same significance, we are capable of determining those factors which have the most impact. Toward this end we have generated 50000 samples.

3. Results

The FP setting gives us a ranking of the factors for each IT-tool under consideration. For example, Figure 1 shows the indices for the tool “Workflow System”. The factor *Skill level in IT-tools* is responsible for 35% of the experts’ decision for this tool; the second most important factor is *Terminology* with 10% and the third one *Cooperation partners* with 9%.

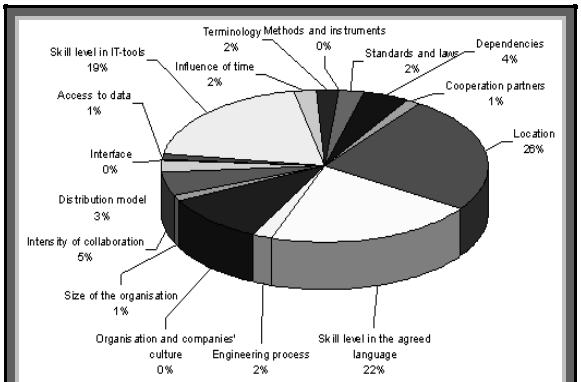


Workflow System

Input Factor	Index	Input Factor	Index
Cooperation partners	0,0836	Interface	0,0566
Location	0,0189	Access to data	0,0215
Level in the agreed language	0,0073	Skill level in IT-tools	0,3003
Engineering process	0,00000241	Influence of time	0,0475
Organisation's culture	0,0019	Terminology	0,0896
Size of the organisation	0,0734	Methods and instruments	0,016
Intensity of collaboration	0,0582	Standards and laws	0,0298
Distribution model	0,0161	Dependencies	0,0699

Figure 1. Factor prioritisation for the tool "Workflow System"

However, another example clearly shows that this factor prioritisation is not the same for each tool. For example, for the tool "Online-Conferencing": the factor *Location* is responsible for 26% of the experts' decision for this tool; the second most important factor is *Skill level in the agreed language* with 22% and the third one *Skill level in IT-tools* with 19%.



Online conferencing

Input Factor	Index	Input Factor	Index
Cooperation partners	0,0117	Interface	0,0023
Location	0,2168	Access to data	0,0086
Skill level in the agreed language	0,183	Skill level in IT-tools	0,1704
Engineering process	0,0172	Influence of time	0,0191
Organisations' culture	0,0895	Terminology	0,0214
Size of the organisation	0,0111	Methods and instruments	3,03E-07
Intensity of collaboration	0,0416	Standards and laws	0,0213
Distribution model	0,0235	Dependencies	0,0387

Figure 2. Factor prioritisation for the tool "Online-Conferencing"

These two examples demonstrate that each tool has its own characteristics and that their implementation doesn't depend on the same boundary conditions of global development projects.

Subsequently, each factor doesn't have the same influence on each tool. For example, in Figure 1 the factor *Skill level in the agreed language* only influences the selection of the tool "Workflow System" with 1 %, whereas it influences the choice of "Online-Conferencing" with 22%. Such considerations substantially aid project and IT-managers in decision-making regarding the right IT-support-platform for global distributed development projects. In fact, to evaluate whether a tool is appropriate for their project or not they have the possibility of considering only the most relevant factors (e.g. factors having more than 5% of influence) instead of all the 16 factors, which would be more time-consuming. E.g., to evaluate whether an "Online-Conferencing" tool would be appropriate for the support of a specific project, the consideration of 4 factors as opposed to 16 is sufficient: *Location*, *Skill level in the agreed language*, *Skill level in IT-tools*, and *Intensity of collaboration*. The other way around, if they already have analyzed their project characteristics, they can very quickly eliminate some IT-tools, since they know which conditions of the influencing factors are not fulfilled.

4. Application of the results to a real industrial global development project

A cooperation project between BMW and another Original Equipment Manufacturer (OEM) needs a set of appropriate IT-tools to support their information, communication, and coordination processes. We want to use the results presented in section 3 to propose a set of IT-tools. Toward this end we will first describe the project with respect to the factors presented in Table 1.

Each of the main cooperation partners has, of course, suppliers involved in the development process, but they are not involved together at the global cooperation level. The suppliers only have bidirectional interfaces with their respective OEM.

Subsequently, the two partners are from different companies, each of which has a very marked culture, belongs to the category of large-scale enterprises and is located either in Germany, in England (by BMW) or in France. Their level in the agreed project language is more or less good.

The participants are integrated in SE-Teams (Simultaneous Engineering) and work on their tasks parallely. This is in fact one of the main characteristics

of SE-Teams. For the same reason the collaboration is substantially integrated into the daily work schedule. The tasks are equally distributed between both OEMs.

Organisational as well as technical interfaces are quite striking. In fact, the partners are competing enterprises, a circumstance, which complicates their relationship, flexibility and facilities. The technical and organisational ability to access the relevant data is, therefore, also very restricted.

The skill level in the use of IT by the participants is quite low. It is not in fact their specialty, and very few participants are really capable of dealing with IT.

The development of the motor is a single project and considered to be time-critical, because of possible repercussions on the companies' profits.

The vocabulary employed by the participants varies since each of them is a specialist in a different domain. They also use different methods and instruments depending on their respective domain and their mother company. However, they follow the same industrial standards. The dependence of the participants on their original company is very high due to the tight organisations' culture.

Let us first consider the factor *Skill level in the IT*. This factor inherently eliminates 4 of the 18 IT-tools under consideration: Workflow System, Change Management Tool, Electronic Meeting System, and Knowledge Management System. Taking the factor *Terminology* also into consideration eliminates 4 more tools. Repeating this step again refines our selection one more time. In a very short period the tools that come into consideration are left over: Electronic Calendar, Online-Translator, Project Management System, and E-Mail. Hence, we are able to sort out quickly a set of IT-tools appropriate for specific project configurations and merely only with the results of the sensitivity analysis.

5. Conclusion

In global development projects communication, coordination, and cooperation processes between participants are technically difficult to support as well as being time consuming. Information and communication systems help such projects to become successful. However, choosing the most appropriate IT-support can be quite difficult since many influencing factors must be considered and the defining of precise requirements on collaboration software proves to be extremely complex.

To simplify this decision the paper presents the results of a sensitivity analysis on influencing factors for global development projects; this analysis showed that some influencing factors have much more impact on the selection of a specific IT-tool than others. Moreover, it is not always the case that the same factors have the same impact on the selection of one tool or another. Each tool has its own characteristics and its implementation doesn't depend on the same boundary conditions of global development projects.

These findings can simplify the decision-processes of project and IT-managers considerably. It aids project- and IT-managers in decision-making regarding the right IT-tools for global development projects substantially and thus influences the phase in which the user requirements of collaborative-software are defined. In fact, users often want to collaborate, but do not know how to go about this. The classification of implementation's conditions regarding each IT-tool aids the project leader in defining the requirements for the collaborative-software to develop. Moreover, with the help of the classification, the evaluation of a tool only requires the consideration of its relevant factors instead of carrying out a time-consuming analysis of the entire project which is in need of support. It is of great significance for time-reduction processes in the early phases of development projects, especially if the project must get started quickly, and the managers lack time to wait for an analysis of the project's configuration and possible IT-support.

6. References

[1] Grudin, J., "Computer-supported cooperative work: Its history and participation", *IEEE Computer 27(5)*, 1994.

[2] Ellis, C. A., Gibbs, S. J., Rein, G., "Groupware: Some issues and experiences", *Communications of the ACM*, 34(1), 1991.

[3] Laurent, C., "Design of IT-Collaboration-Platforms with Fuzzy Logic", *Current Research in Information Sciences and Technologies Multidisciplinary approaches to global information systems*, Open Institute of Knowledge, Merida, Spain, 2006.

[4] Anderl, R., Lindemann, U., Thomson, B., GAUL, H.-D., Gierhardt, H., Ott, T., "Investigation of Distributed Product Design and Development Processes", *Proceedings of the 12th International, Conference on Engineering*, 1999.

[5] Gierhardt, H., Gaul, H.-D., Ott, T., "Distribution in Product Design and Development Processes", *Proceedings of the 1999 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, 1999.

[6] Gierhardt, H., *Global verteilte Produktentwicklungsprojekte – Ein Vorgehensmodell auf der operativen Ebene*, München, Dr. Hut 2001

(Produktentwicklung München, Band 46), TU München, 2001.

[7] Hartmann, Y., *Controlling Interdisziplinärer Forschungsprojekte*, 1998.

[8] Laurent, C., "Analysis of the Survey: IT-Platforms for Cooperation Projects", *Technical Report*, TU München, 2007.

[9] Saltelli, A., "Global Sensitivity Analysis: An Introduction", *Proc. 4th International Conference on Sensitivity Analysis of Model Output (SAMO '04)*, Los Alamos National Laboratory, 2004.

[10] Saltelli, A., Chan, K., Scott, E.M., "Sensitivity Analysis", *Probability and Statistics series*, John Wiley & Sons publishers, 2000.

[11] Saltelli A. Tarantola S., Campolongo, F., Ratto, M., "Sensitivity Analysis in Practice. A Guide to Assessing Scientific Models", *Probability and Statistics series*, John Wiley & Sons publishers, 2004.

[12] Wagner, S., *Cost-Optimisation of Analytical Software Quality Assurance*, Technische Universität München, 2007.

[13] SimLab 2.2., http://webfarm.jrc.cec.eu.int, 2005.

Requirements Management Infrastructures in Global Software Development
Towards Application Lifecycle Management with Role-Oriented In-Time Notification

Matthias Heindl, Franz Reinisch
Support Center Configuration Management
Siemens Program and Systems Engineering
Vienna, Austria,
matthias.a.heindl@siemens.com

Stefan Biffl
Institute of Software Technology
and Interactive Systems, TU Wien,
Vienna, Austria,
stefan.biffl@tuwien.ac.at

Abstract

Global software development can be characterized by globally distributed project teams (e.g., project management, development and test teams). Each of these teams has requirement-related tasks (e.g., developers implement requirements, testers test requirements) and workflows that need tightly integrated tool support (requirements management, IDEs, test management).
In this paper we describe from the perspective of a large GSD software development company: (a) major requirements management needs in GSD, (b) current approaches for requirements management in GSD and their insufficiencies;(c) an initial approach to improve these insufficiencies; and (d) further research steps to better address the needs based on a common data exchange format between development/management tools and a role-oriented notification system.

1. Introduction

Siemens Program and Systems Engineering (PSE) is a research and development entity within the Siemens group with more than 6,000 employees. PSE software and systems development projects cover a broad range of application domains, such as: telecommunication and information technologies, automation and control, power, transportation, medical solutions, components, and space technology. In recent years Siemens PSE increasingly started globally distributed development (GSD) projects supported by new permanent offices in Eastern Europe, Turkey, and China.

We characterize a highly distributed development project as a project with team members in 2 or more countries. Such projects typically exhibit the following characteristics: 1. By definition not all team members can work at the same location; travel delay between the project headquarter and other locations is significant, e.g., more than 2 hours. 2. Thus, there is less opportunity for flexible direct (face-to-face) communication: regular meetings occur less often than bi-weekly; *ad hoc* meetings are very hard to achieve due to the traveling delay or high costs to establish meetings, e.g., considerably more than 1,000 Euro even for a short meeting [5].

The Support Center Configuration Management (SCCM) unit provides PSE-internal consulting for employees and facilitates experience exchange among development teams on core topics like configuration and requirements management. PSE Support Centers address success-critical topics to (a) bring employees with similar problems together, and (b) support communication of experience that could help to solve a current problem. There are multiple other Support Centers like project management, configuration management, usability, and testing. Over the years, SCCM staff has gained experience on needs in GSD projects for requirements management tool support and on strengths and limitations of currently used approaches.

We report on the requirements management needs in GSD projects based on data that the SCCM collected from project participants during our consulting and support activities for these projects. Major needs are: timely and effective information exchange; cheap requirements tracing across tool borders; and of tool integration that allows retrieving data from one tool, e.g., a test management tool, and display it in another tool, e.g., a requirements management tool.

The remainder of the paper is structured as follows: Section 2 summarizes needs for automating requirements management in GSD projects compared to one-site projects. Section 3 reports experiences with current approaches to manage requirements in GSD projects. Section 4 proposes the concept of tool-supported role-

based notification to provide in-time the right level of information to the roles involved in a work task. Section 5 summarizes the contributions of the paper and suggests further research work.

2. Requirements Management Needs in GSD projects

In GSD projects typically multiple distributed teams work on the realization of requirements: e.g., project management (at site A), development teams (at sites B, C), and test teams (at site D) [4]. Typically these teams have many requirements-related tasks: project managers deal with the specification of requirements, change management, and requirements traceability; developers develop source code based on the provided set of requirements and report progress to project management: which requirements are already implemented. Testers create test cases for requirements, execute test cases, and report tests, e.g., which test cases were completed successfully and thereby which requirements have been covered so far by testing

Typically, these roles use tools that facilitate their work. A typical tool set in such projects may consist of a requirements management (REQM) tool, a configuration management tool, development tools (IDE), test management tools, collaboration platforms, and communication tools (like email and *Netmeeting*).

The SCCM frequently receives requests for REQM and configuration management (CM)-related support from GSD projects. During these consulting and support activities and in expert network meetings, where project participants from different projects come together to exchange and discuss current project-related REQM problems, the SCCM frequently receives feedback about REQM needs in GSD. Based on this feedback, we present the following list of main requirements management needs of PSE project participants in GSD projects:

Timely and effective information exchange (requirements awareness). Notification about changes of relevant information, e.g., for change propagation as requirements are likely to change while developers work on their dependent implementation. As a consequence developers may implement outdated requirements, wasting effort on re-work of the code. Information about changes or other events at one site that are relevant for roles at other sites shall be provided timely, to avoid unnecessary rework. Furthermore, this information exchange shall be effective in the sense that each role gets exactly the information it needs (neither "flooded with email" nor "starving for information"), which is a challenge for project management in general but even more in GSD projects due to the limited availability of synchronous communication (geographical distribution across several time zones). Essential in that context is permanent accessibility of relevant information, e.g., a developer implementing a change request (CR), should have easy access to the task description tracing back to the original CR and, if necessary, further back to the documentation of related decisions and contact persons like stakeholders who may provide background information. It is important to maintain a clear picture of responsibilities in the project and dependencies (between artifacts, persons, and tasks) to allow the timely notification of relevant roles. Damian and Zowghi [2] support the importance of such "requirements awareness mechanisms". Collocated teams usually benefit from social mechanisms and processes that facilitate the work practice and diminish the perceived need for explicit requirements awareness activities. However, this kind of access to informal communication is significantly limited in geographically distributed teams [3].

Requirements traceability. Tracing requirements [5] back to their origin or having rationale for them helps to better understand the meaning of requirements. In GSD projects traceability is even more important than in collocated projects, because requirements cannot be clarified easily during informal "corridor talks". Furthermore, captured traces between requirements and other artifacts like source code elements and test cases give the project manager hints for progress monitoring ("is requirement A already implemented and tested?"), change impact analysis ("which artifacts do I have to change when requirement B changes?"), or coverage analysis "(is every requirement sufficiently covered by test cases?").

Integration of tools. Comprehensive tool support is needed to enable consistent, error-free, and up-to-date information exchange, requirements awareness, and traceability in a GSD context. Tool support mostly consists of tool sets (requirements, development, and test tools) that can interact in principle providing the basis for redundancy-free, consistent storage of data and exchange of data between tools (via tool interfaces). However, tighter tool integration than provided is needed, e.g., for project managers who want to see certain parts of test data from within the requirements management tool, so they do not have to use the test management tool itself.

In summary, tool support for requirements management in GSD projects has to meet the following success criteria: (1) permanent access to relevant information, e.g., history of a requirement; who worked on which requirement when, which decisions were made why (requirements awareness); (2) timely notification of relevant role on occurrence of specified events and conditions, e.g., changes to some requirements or dependent documents; (3) easy means to facilitate requirements tracing across tool borders as a prerequisite to manage dependencies; and (4) the right, i.e., higher than currently available, level of tool integration.
In the following section we will outline current approaches for requirements management and their shortcomings.

3. Current REQM Approaches in GSD

Currently the major approaches for requirements management (REQM) in GSD projects can be sketched as: a) point-to-point integrations of tools (instead of an homogeneous tool platform), b) lots of telephone calls and c) lots of emails. Based on SCCM's experiences, this section provides an overview on REQM approaches and outlines our work-in-progress to improve these approaches. The SCCM's knowledge about REQM in GSD projects comes from their consulting and support services (tool setups and customizations) for some 100 projects per year, most of them highly distributed.
From our point of view, the basic problem of REQM in GSD projects is mostly insufficient integration of data between development and management tools. REQM is mainly is a communication task (requirements communication [8]), e.g., informing people about requirements changes. However, fast and effective information exchange is not well supported by the tools used routinely (e.g., *Requisite Pro*, *TestDirector*, *Eclipse*, *Sharepoint*). As a result, data kept in different repositories get inconsistent, which hinders project managers, developers, and testers to achieve and maintain a consistent view on dependencies between artifacts.
While lots of interesting information exists in these tools, the weak integration makes it hard to communicate this information and data to interested project participants by way of tool infrastructure. Therefore, project managers and others have to exchange information, e.g., notifications about change requests or identified bugs, by external communication tools, such as telephone, email, and *Netmeeting*. This communication effort can be significant: a GSD project manager may often have telephone conferences which take hours with all relevant sites for a current issue. Furthermore, there is the risk that sometimes people simply forget to notify other roles about important change events.
With a proper tool infrastructure that provides access to relevant information across tool borders and timely notification in case of important events, we see the possibility to the reduce costs for communication, avoid a significant part of rework, and mitigate the risk of non-communication: on an average project overall effort could be lowered by up to 10% and variability of project cost and quality significantly reduced.

The currently available point-to-point integration mode between tools allows only limited data exchange and does not provide notifications for relevant users in case of important events. These integration approaches also introduce new challenges [11]:

- For each new tool that should be used in the project, new integration instances have to be built to all the existing tools, which is often difficult to establish and costly to maintain.
- A single tool for each role. The problem with role-based tools is that roles are anything but uniform, varying by company, by business unit, by development team, and even by individual. When a customer's role set (e.g., analyst and architect in a combined role) does not match the roles for which vendors have built tools (e.g., a requirements management tool that does not provide modeling facilities), the IT organization has to choose between changing its roles, licensing multiple tools for a single role, or purchasing more features than a given role is likely to need. To provide an audience of diverse practitioners with all the features they need, vendors end up stuffing tools with so many features that they get very hard to use in practice, e.g., a configuration management (CM) tool that can be customized for multiple purposes, may be overly hard to use. The results are complex and expensive tools that have more functionality than any individual is likely to need, which may hinder potential users to use the tool most effectively and efficiently.
- Redundant features locked in practitioner tools, e.g., each tool has its separate user management system, which incurs extra effort for tool administrators to keep privileges up-to-date and consistent.

In order to address the limited data exchange possibility of available tool integrations, which is a main hurdle to fulfill the needs identified above, we have, as a first step, developed an *Eclipse* plug-in that provides an interface between a requirements management tool

(*Requisite Pro*) and *Eclipse* to allow developers easily to trace requirements into source code within their IDE and without the need to access an external requirements management tool. The plug-in serves as a basis to evaluate how integrations between tools can be improved. The plug-in provides the following features [6]: The tool support automatically imports a list of requirements from *Requisite Pro* into *Eclipse*. Thus, traces between requirements (requirements information is displayed directly in *Eclipse*) and source code can be created in the developer's environment (*Eclipse*) without the necessity to open other tools or look up references in requirements documents. Developers can create a trace via selecting a requirement from the requirements list. The traceability information is automatically re-imported into *Requisite Pro*, where the project manager can view it as a traceability matrix.
Clear benefits of this approach are:

- Significantly reduced effort for tracing;
- Facilitation of tracing as unobtrusive part of developers' usual work practices (process-driven tracing);
- Reduced mental efforts for correctly selecting traces from a list, which is on average much less error-prone than traditional manual tracing;
- Avoid making users use extra tools for tracing.

The plug-in is an approach to coordinate activities between requirements engineering or project managers, who work on the requirements in the requirements management tool, and the developers working with the IDE. Furthermore, the plug-in also facilitates a change notification system: when a developer works on a certain requirement from the requirements list in *Eclipse* and the project manager works on that requirement in the requirements management tool, e.g., changes the requirement, the plug-in can propagate this change to the IDE and highlight the particular requirement. Thereby, the developer is notified and can clarify the requirements state. Thus, he can avoid risky work on potentially outdated or inconsistent requirements.

Our initial experiences with the plug-in-based requirements tracing approach indicate a significant systematic influence on requirements tracing (RT) in larger projects. E.g., a daily reduction of trace efforts from 30 down to 3 to 5 minutes is likely to result in significantly higher tracing acceptance and hence more detailed daily information for the project manager, compared to expensive manual tracing in bi-weekly cycles. Due to the positive experiences with the plug-in, namely cheaper and easier tracing across tool borders, information exchange (exchange of requirements data and traceability information, and display in both tools), and change notifications, we want to extend this approach to other tools, e.g., test management tools, so that requirements data exchange is possible within all tools in the tool set. We follow an application lifecycle management (ALM) strategy [10] to accomplish that goal: concretely, we will in a next step use a common data format to exchange requirements-related data between tools (like in the *Eclipse* plug-in). For this we will reuse the requirements interchange format (RIF) [9], an XML standard for requirements data exchange from the automotive area. Based on RIF, we will build interfaces that support data exchange between the tools via RIF. By adapting (and maybe extending) RIF we can then provide "richer" tool integrations that enable requirements awareness.

4. Role-Oriented Notification System

In order to fully address the REQM needs GSD project participants need a generic mechanism to ensure timely notifications about changes to requirements or other artifacts in addition to the common data exchange format mentioned above [13][14]. Multiple requirements-related events may happen concurrently at several sites across a GSD project (see overview in Table 1), which would seem relevant for project team members at other sites. For example, when requirement 4711 in the requirements management tool changes (event in the requirements management tool), the developer who works on this requirement can be notified, e.g., by displaying and highlighting the requirement in the IDE (notification).
Another scenario is build automation: there is a tester that monitors test execution. If a test run fails, the project manager, who needs information about test progress, and the responsible development team, which will have to correct the bugs, need to be informed. Instead of forcing the tester to actively inform the project manager and developers by telephone or mail, it would be much cheaper if the tool infrastructure notifies them automatically, e.g., send the project manager a notification within his requirements management tool and let him view test results there, too. The developers get a notification with in their IDE and can view failed test cases in order to correct the relevant code.

Notifications are currently often triggered by persons, e.g., via telephone calls or emails. As explained above, this way of notification is often costly and unreliable; effective tool-based notifications promise to reduce the communication effort. Of course, the concept of notification is not new, e.g., there are already database triggers that implement some form of notifications. A nov-

elty is the integration of events coming from heterogeneous systems as input to a rule engine that can correlate, aggregate, and filter events in order to provide triggers for relevant role-oriented notifications.
Based on the implementation of a common data exchange format, which can form a tool integration bus, we develop a notification system that can assist project users in defining useful notifications across GSD sites. The notification system should be role-based, so that a user can define which kinds of notifications are relevant for him. As main challenges we see:

- Provide useful notifications (correct, helpful; e.g., receiver would be willing to pay for getting a certain kind of message);
- Avoid information overload with irrelevant or wrong notifications;
- Keep an overview on the correctness of a large rule set; e.g., which rules should be active.

Events in the REQM tool
new requirement is inserted
Existing requirement is changed
New user is added
User privileges are changed
Events in the IDE
Trace is created between source code element and requirement
Events in the Configuration Management tool
New change request was submitted
State of an artifact changed
Check-in of an artifact
Events in the test management tool
Bug report is inserted
Test cases for a particular requirement are reported as successfully tested.

Table 1. Examples for events in GSD tool support.

Notification can work based a rule engine, e.g., a "correlated events processor" (CEP) [7][12], which is connected to the set of tools used in a project via a common "tool integration bus". Figure 1 illustrates how the used tools are connected via that bus. The rule engine receives events from the other tools and generates notifications for relevant roles/users via the tools used (eventually an ontology can be help to bridge the languages of the different system users). Thereby, a good part of notification efforts are shifted from the users to the tool infrastructure.
A role-based notification system should provide three types of information: (1) *artifact information*: accurate information about the current state of a document; (2) *event information*: when an event occurs (e.g., change to a document) relevant roles can be notified by the system; (3) *artifact history*: events are stored in a database and related to artifacts so that users can search the database for events that happened to requirements and other documents in the past.

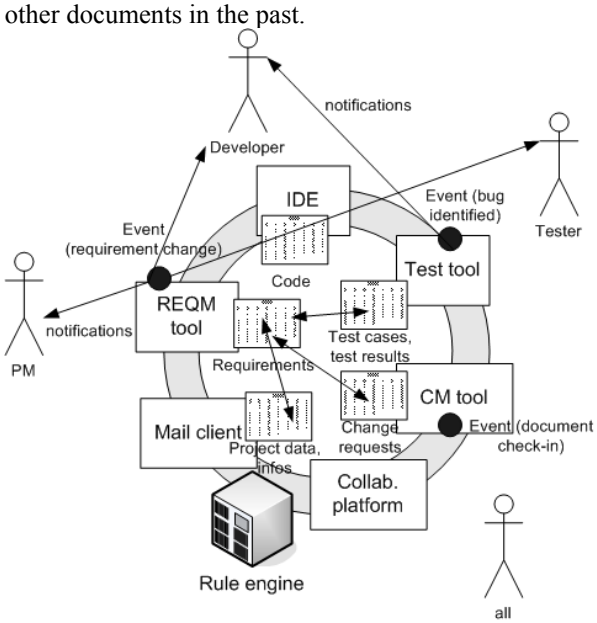


Figure 1. Role-oriented notification system.

We plan the following next steps to implement such a role-oriented notification system:

1. Definition of notification rules based on a syntax to formulate rules that determine when to trigger a notification: Notification rules describe <*whom*> (list of persons or roles) to notify <*in what way*> (e.g., e-mail, SMS, entry in change log) <*when*> (e.g., immediately; batch every hour/day) due to <*what change*>. *Change* can be an observable event or state change regarding an artifact or project state, e.g., some expected event did not happen during the specified time window. (see further examples in Table 1). An SQL-like example rule for the build automation scenario in section 3 could be:

```
if TESTRUN 0815 FAILS NOTIFY (PM, Peter
Mayer) BY (ReqPro, IDE) and SEND FAILED
TEST CASES.
```

The rule says: in case of an erroneous test run the project manager (whoever has this role) should be notified via *Requisite Pro*, and Peter Mayer (a developer) gets notified via his IDE. The "send" option defines which data to send with the notification, e.g., the failed test cases as info for the PM and the developers, so that they know, which code to check.

2. Escalation. If a condition in the rule set indicates a state that the system cannot handle automatically, the

default is to escalate the issue with appropriate context information to a role that is expected to have enough overview and competence to provide a reasonable decision.

3. Tool support. Tools can support role-oriented notification by interpretation of machine-readable dependency information (that would be cumbersome for humans to follow). In a next step we want to use the CEP to enable notifications. CEP correlates related events to efficiently filter "interesting" events as basis for notification. Recent notifications can be stored in a database, so users can choose whether to

- receive change notifications immediately or summarized in regular time intervals;
- look at recent changes in the change database;
- look at the current version of artifacts (if there are many changes and/or major structural changes).

The CEP receives events from the tools, processes them according to a rule set and sends notifications to users via the tools. The processing of the events is based on user-defined notification rules. Each user can define for which events he wants to be notified. The challenge is to configure the CEP so users can get the most relevant [1] information in a timely manner.

5. Conclusion and Further Work

The Support Center Configuration Management (SCCM) collected data from GSD projects about requirements management needs: timely and effective information exchange, requirements awareness, requirements traceability across tool boarders, and comprehensive tool integrations. Currently, these needs are only weakly fulfilled, due to insufficient tool integrations and notifications on important events, e.g., requirements changes, whose delivery mainly depends on team members.

In this paper we outlined requirements management needs in GSD, as collected by the SCCM and illustrated current approaches for requirements management in GSD. Furthermore, we proposed a requirements tracing plug-in, which systematically improves data exchange, tracing, and notification facilities of currently available tool integrations for significant impacot on trace quality in a project. Next research steps are to put the ideas of the plug-in on a firm footing by: (1) developing a common data exchange format, so requirements information and related data can be viewed in the tools used in the GSD project, (2) designing and prototyping a role-oriented notification system.

By implementing both, a common comprehensive data exchange format and the role-oriented notification system, we can strongly improve currently used REQM approaches by providing: (1) flexible information exchange between used tools (2) timely tool-based notifications which disburdens the project participants from actively notifying others, (3) improved traceability across tool borders. Expected systematic effects on GSD projects are to increase the speed and reliability of important parts of project communication and to reduce risks coming from inconsistent data stored in different tools.

References

[1] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grünbacher, "Value-Based Software Engineering", *Springer*, 2005

[2] D. Damian, D. Zowghi, "Requirements Engineering challenges in multi-site software development organizations", *Requirements Engineering Journal, 8, pp. 149-160*, 2003

[3] D. Damian, J. Chisan, P. Allen, B. Corrie, "Awareness meets requirements management: awareness needs in global software development", *GSD workshop, International Conference on Software Engineering (ICSE)*, 2003

[4] J. D. Herbsleb, D. J. Paulish, M. Bass, Global software development at siemens: experience from nine projects, *International Conference on Software Engineering*, 2005

[5] M. Heindl, S. Biffl, "Risk Management with Enhanced Tracing of Requirements Rationale in Highly Distributed Projects", *ICSE, GSD Workshop*, 2005, Shanghai.

[6] M. Heindl, F. Reinisch, S. Biffl, "Integrated Developer Tool Support for More Efficient Requirements Tracing and Change Impact Analysis", *Technical Report – Vienna University of Technoloy*, January, 2007

[7] D. Luckham "The Power of Events" Addison Wesley, 2002

[8] V. Mikulovic, M. Heiss, "How do I know what I have to do? the role of the inquiry culture in requirements communication for distributed software development projects", *ICSE 2006*: 921-925

[9] http://www.automotive-his.de/rif/doku.php

[10] C. Schwaber, "The Expanding Purview of Software Configuration Management", *Forrester Research*, July, 2005

[11] C. Schwaber, "The Changing Face of Application Life-Cycle Management", *Forrester Research*, August, 2006

[12] www.senactive.com

[13] de Souza, et al., "Supporting Global Software Development with Event Notification Servers", International Workshop on Global Software Development, 2002

[14] de Souza, et al., "Management of Interdependencies in Collaborative Software Development", International Symposium on Empirical Software Engineering, ISESE 2003

Communication Tools in Globally Distributed Software Development Projects

Tuomas Niinimäki
Helsinki University of Technology
Software Business and Engineering Institute
P. O. Box 9210, FIN-02015 TKK, Finland
tuomas.niinimaki@soberit.hut.fi

Abstract

This paper presents a research proposal on globally distributed software development projects. The research studies the communication tools in such projects, the practices, challenges and problems associated with using these tools in globally distributed software development. The research method is a multiple case study of three software companies. The data will be collected with interviews and questionnaires from the software development practitioners. The proposed research methodology is based on action research.

1 Introduction

The overall trend in software companies since the 1990s has been to outsource and offshore software development. The main driver for this development has been the significantly lower software development costs in less developed countries. In many cases, however, also other factors contribute to this trend, such as the availability of trained workforce, necessity of getting closer to customers and foreign markets, and the possibility of around-the-clock development. [6, 9, 3]

Distributed software development does not come without cost. Working on software products even on collocated teams is difficult due to the properties of computer software itself, such as its invisibility, constant change, ambiquously defined scope and the large amount of tacit information on the software product and development process. Companies and teams developing software face many problems and challenges during software projects, many of which are not of technical nature. It is also known that software development involves a large amount of communication and collaboration between different stakeholders and within the software development team itself. [6, 8]

Furthermore, in software engineering, the software processes play an important role, as every organization has a slightly different view on how to make the software, and there are no standardized processes [1]. In addition to describing the software product itself, the software process must often be communicated. There are also commonly interdependencies between various tasks and activities in software development projects [9]. The amount and importance of tacit knowledge thus becomes even larger in software projects than it is in more traditional engineering projects. The issue with tacit knowledge comes even more important when cultural differences are larger: there can be completely different views about the practicalities of work as well as different level of expectations within the parties of globally distributed software development project [2, 4].

Even while it is widely understood, that communication is important, and in most cases the main source of challenges in distributed projects, communication is rarely disciplined as rigorously as many other aspects of software engineering. We argue, that this is caused by not identifying the need for both formal and informal communication, as well as the specific needs caused by geographical and time zone distance. Communication planning is also difficult due to lack of understanding in communication tool properties.

This paper describes a proposed research focusing on communication challenges in globally distributed software development projects. We study the current state-of-the-practice in selected software projects from three software companies, and evaluate their communication challenges, and the tools and practices used to solve these challenges. Based on this and the literature survey for the state-of-the-art communication tools and practices, we actively collaborate with the practitioners using the action research methodology to improve the communication process in the studied projects.

2 Previous work

The proposed research builds on previous research conducted by Maria Paasivaara in her doctoral dissertation [10]. She studied 12 projects from manufacturing and software development industries. Her research identified eighteen

successful communication practices used in distributed collaboration between organizations.

Communication is very important in software engineering, and especially so in distributed software development [6]. Globally distributed software projects use many different communication tools. These tools include e-mail, instant messaging, telephones, teleconferencing, video conferencing, groupware applications, version control systems, integrated development environments and shared workspaces [2, 5, 3, 13, 12]. These communication tools can be classified in several ways. Classification can be based communication content type: some tools provide only one media type, while others may combine audio, video and text [13]. Synchronism is another property, that affects the tool use: e-mail is asynchronous in comparison with instant messaging, and thus less suitable for tasks requiring intensive interaction, but more suitable for communicating across time zones [13, 7, 1]. There is, however, very little empirical data on how and when these communication tools are actually used in the industry.

A survey on global software development companies studied the problems and challenges in distributed projects. The study classified problems into eight categories. It was found that 74% of the problems in distributed projects were caused by "communication and contacts". These problems were caused by two major factors: cultural differences and physical distance. Problems caused by cultural differences include language difficulties in this study. It was often the case, that language skills were so low, that fluent conversation became impossible, most notably in teleconferences. The survey reports, that 30% of the problems in studied projects were related to "communication tools". This was the problem category receiving the least responses. However, there were a number of difficulties in using especially video conferencing, that were reported in the study, but were not included in "communication tools" category. The reason for this was not reported. [8]

It can be argued, that a number of problems that were reported as "non-tool" -related, could be solved by appropriate use of communication tools. Even while the challenges related to the technology, implementation and availability of communication tools suitable for global collaboration have been mostly solved, the adoption and use of communication tools deserve more attention and care. This requires understanding the limitations and specific properties of selected communication tools [7]. Earlier research has reported that there is lack of communication tools designed especially for global software development projects [3], but it can be argued whether this still is the case, as the communication technology has advanced greatly since the research was done. Even though the technology is available and easier to use, it has been reported that the barriers in communication still include the adoption and setup time for communication tools [13].

It has been reported that software developers spend an average of 75 minutes per day in "unplanned interpersonal interaction" [11]. In general, face-to-face meetings were considered as the best way to communicate [8]. Physical distance reduces the number of face-to-face meetings [6, 8]. E-mail, teleconferences and video conferences are the most common tools used to substitute the lack of face-to-face meetings in distributed projects [8, 1]. The lack of face-to-face communication causes several problems, such as misunderstandings between people in different sites, redundant work to be done in the projects or even work not done at all [8]. In distributed projects, especially the informal communication is reduced significantly. This causes challenges understanding the context in different sites and reduced awareness of work done by distant team members [5]. However, in addition to the lack of communication, different cultural background, different work histories, different educational backgrounds and language barriers affect this issue [5]. It can be argued, that proper application of communication tools and practices could support informal communication, building team spirit and trust as well as leverage the difficulties in understanding, cultural differences and language barriers. The previous research provides very little empirical research on this subject.

Previous work provides a view on the state-of-the-practice and the state-of-the-art communication tools and practices. Previous research however provides very little discussion on how these tools are used in distributed software projects, what are the specific considerations and practices for each communication tool, and can these tools really help in solving the challenges in globally distributed software development.

3 Methodology

The proposed research is a multiple case study on communication challenges on globally distributed software development projects.

The case study is conducted by interviewing practitioners from several globally distributed software projects. These projects are selected in co-operation with the practitioners in software companies, and the selection of projects is based on the experiences gained from the projects so far. The projects to be studied include both successful and challenged projects. Based on the interviews, two surveys are planned to be done to the practitioners participating in the selected software projects.

The main research goals in the proposed research are to identify the tools and practices used in communication in selected globally distributed software development projects, the best practices to use these communication tools, the challenges and problems in the both adoption and use of

these communication tools, and the challenges and problems these communication tools cannot solve.

A communication tool is defined widely in this research: term "communication tool" covers both traditional methods of communication, such as face-to-face meetings, teleconferences and electronical mail, and more novel communication tools, such as three-dimensional virtual environments, virtual collaboration environments and wiki pages. In addition to forementioned tools that can be clearly understood as communcation tools, the research also includes the less obvious means of communication in distributed software development, such as issue tracking and version control systems.

3.1 Research questions

The research to be conducted is based on following research questions:

1. What are the communication challenges globally distributed software development projects experience?
2. What are the communication tools and practices used in globally distributed software development?
   (a) How communication tools are used in globally distributed software development?
   (b) How does the use of communication tools and practices change between various stages and activities of a software project?
3. What are the challenges with communication tools in globally distributed software development projects?
   (a) How do these challenges make the adoption of communication tools more difficult?
   (b) How do these challenges affect the use of communication tools?
4. What are the communication tools and practices that specifically support globally distributed software development projects?
5. What are the challenges and problems these communication tools do not solve?

The motivation for the first research question is to find out the current communication challenges in GSD projects. The second question tries to find out the communication tools and practices used to overcome these challenges in the studied software companies as well as in the industry in general. The combination of first two research questions should provide a view on the state-of-the-practice in the studied software companies. Based on this view, we try to identify the main challenges and problems in the communication and the use of communication tools in selected software projects with the third research question. The fourth research question aims to find out the useful and applicable communication tools and practices which could support the globally distributed software projects. The fifth research question tries to find out the issues current communication tools and practices are not able to handle, even if applied in a proper, disciplined way.

The proposed research can be divided roughly into two stages. The first stage is about collecting the research data and the assessment of current situation in the studied software companies. The first stage of proposed research tries to address the first two research questions. The second stage, addressing the last three research questions, tries to find solutions or conclude the absence of solutions related to challenges on communication tool use.

3.2 Collecting research data

These research questions will be answered based on both a literature survey on previous research, and interviews and questionnaires on participating software companies.

The research will be conducted in several software companies. A set of projects will be selected from these companies co-operatively with the representatives of respective companies. The proposed research uses interviews and questionnaires to collect research data. The selected key personnel of these software projects, both the in-house and the offshore, will be interviewed. The interviews are semi-structured, they are recorded and notes are made during the interview session.

The interviews are made in several rounds. The first round of interviews is aimed at drawing an overview of the projects, to gather information about the currently used communication practices and their applicability, and the current challenges and best practices in communication. Consecutive interview rounds are aimed to focus on some of the communication practices in use, and to study their use during the project.

Together with the interviews, the proposed research contains action research cycles. During these cycles new communication tools and practices can be introduced to the teams, and the experiences and evaluations of these practices are gathered during the interviews.

The interviews will be analysed qualitatively. The written notes from the interviews are read through, and the material is arranged into categories. During the first round, the analysis will focus on gathering information on the projects and their members, but consecutive interview rounds concentrate more on the actual communication tools and their use in the globally distributed software projects.

The interviews are accompanied with questionnaires,

which will be introduced after the first interview round. The questionnaires will be used to gather quantitative and qualitative data from the organizations, including the frequencies of use of various communication tools and practices, the attitudes towards them and their fitness for activities of a software project. The results from the analysis of this data will be used further to focus the qualitative research to pinpoint the most relevant phenomenon related to the communication tools and practices in the studied software projects.

3.3 Conducting the research

The research methodology builds on action research. We aim to foster active participation and collaboration with the researchers and the practitioners from the software companies. As the research is organized to be composed of several rounds, we seek to give feedback to the software companies during the research project, not only at the end of it. The feedback we aim to give to the companies includes the analysis of their current situation and the main challenges in their globally distributed software projects as well as some solutions for these challenges. Especially during the consecutive interview rounds we aim to introduce new communication practices and tools as seen necessary, and to enhance the use of existing tools. This work will be done in collaboration with the practitioners, based on the research findings as well as experience from the software companies.

To support and facilitate this collaboration, we aim to design the questionnaires and further interview rounds based on the findings from the studied projects as well as based on the issues and aspects identified during the interviews and discussions with the practitioners. We aim to use both the interviews and questionnaire results as a base for discussion and research planning. The discussion within and between the participating software companies will be facilitated in workshops, which we aim to arrange twice a year.

In addition to these results, this research will produce a doctoral dissertation, a number of scientific publications and a workbook.

3.4 Schedule

The proposed research can be seen as two stages: first stage being identified by research questions 1, 2 and 3, and the second stage by research questions 4, 5 and 6. The time dimension on these two stages is interleaved due to the number of studied projects, but it can be fairly assumed, that only answering the first three research questions in fairly complete extent enable the research on the latter three research questions.

The timeframe for the MaPIT research project is three years, spanning from January 2007 to December 2009. The preliminary schedule for the proposed research is shown on

Table 1. Schedule

Date	Activity
01/2007	Initiation of the research project
03/2007 - 09/2007	First interview round
05/2007 - 11/2007	Analysis of first interview round
01/2008 - 04/2008	Second interview round
03/2008	First survey
06/2008	Stage 1 research data collected
09/2008 - 11/2009	Consecutive interview rounds
03/2009	Second survey
12/2009	Stage 2 research data collected

Table 1. The proposed research will be made as a part of MaPIT project, and thus its schedule follows the schedule of the research project. The first interview round is expected to be finalized in September 2007. Consecutive interview rounds are planned to be conducted in cycles of six months. The questionnaire round will take place after sufficient research data is collected on the current situation of the participating software companies. This may be finalized by the spring 2008 earliest. The interview rounds are accompanied with active collaboration between the researchers and the practitioners in action research cycles. The actual schedule for this is to be determined later in collaboration with the practitioners.

4 Potential findings

First ("*What are the communication challenges globally distributed software development projects experience?*") and second ("*What are the communication tools used in globally distributed software development?*") research question will be answered based on the first interview round. Based on the analysis of the interviews, we will draft a communication tool / software project activity matrix. This matrix will contain the available communication tools on one axis and the activities mandated by the software process on the other axis. Consecutive rounds are also used to further refine and add more detail to the communication tool/software process matrix drafted after the first

interview round.

Consecutive interview rounds as well as a questionnaire round are used to address the research question "*How communication tools are used in globally distributed software development?*". For each project we study in the proposed research, we will select a number of communication tools used in the project. The selection of tools will be done based on analysis which tools seem the most important and relevant in the project work, as well as the tools that project personnel has challenges with.

For these tools, we will assess their current use in the project, and the attitudes towards using the tool as well as experience on the tool suitability for different activities in the software project. We will also conduct a quantitative research on user experience about the communication tools and their use in the projects. This research is done by sending out the questionnaires to the personnel of studied software projects. Based on this research data, we aim to answer to research question "*How does the use of communication tools change between various stages and activities of a software project?*".

Research questions three "*What are the communication tools that should be used in a globally distributed software development project?*" and four "*What are the challenges with communication tools in globally distributed software development projects?*" aim to find out the challenges related to the communication tools found in the first three research questions. The answers to these questions should also benefit the participating software companies to improve their software development processes and especially the use of communication tools.

The answer to fifth research question "*What are the challenges and problems these communication tools do not solve?*" is used both to evaluate and verify the research on other research questions as well as motivate further research.

References

[1] R. D. Battin, R. Crocker, J. Kreidler, and K. Subramanian. Leveraging resources in global software development. *IEEE Software*, 18(2), 2001.

[2] M.-C. Boudreau, K. D. Loch, D. Robey, and D. Straub. Going global: Using information technology to advance the competitiveness of the virtual transnational organization. *Academy of Management Executive*, 12(4):120–128, 1998.

[3] C. Ebert and P. DeNeve. Surviving global software development. *IEEE Software*, 18(2), 2001.

[4] R. Heeks, S. Krishna, B. Nicholson, and S. Sahay. Synching or sinking: Global software outsourcing relationships. *IEEE Software*, 18(2), 2001.

[5] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6), 2003.

[6] J. D. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2), Mar-Apr 2001.

[7] L. Kiel. Experiences in distributed development: A case study. *International Workshop on Global Software Development*, 2003.

[8] S. Komi-Sirvio and M. Tihinen. Lessons learned by participants of distributed software development. *Knowledge and Process Management*, 12(2):108–122, 2005.

[9] A. Mockus and J. D. Herbsleb. Challenges of global software development. *METRICS, Proceedings of the 7th International Symposium on Software*, pages 182–184, 2001.

[10] M. Paasivaara. *Communication Practices in Inter-organizational Product Development*. PhD thesis, Helsinki University of Technology, Software Business and Engineering Institute, 2003.

[11] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

[12] V. Sinha, B. Sengupta, and S. Chandra. Enabling collaboration in distributed requirements management. *IEEE Software*, 23(5), 2006.

[13] M. R. Thissen, J. M. Page, M. C. Bharathi, and T. L. Austin. Communication tools for distributed software development teams. *SIGMIS-CPR 07*, 2007.

A Groupware System for Distributed Collaborative Programming: Usability Issues and Lessons Learned

Crescencio Bravo, Rafael Duque, Jesús Gallardo, Javier García, Pablo García
CHICO Research Group. Department of Information Systems and Technologies
College of Computer Science Engineering. University of Castilla – La Mancha (Spain)
{Crescencio.Bravo, Rafael.Duque, Jesus.Gallardo}@uclm.es; javintx@hotmail.com;
Pablo.Garcia@uclm.es

Abstract

The advances in network and collaboration technologies enable the creation of powerful environments for collaborative programming. This article describes one such environment, called COLLECE, primarily used for collaborative learning. This system supports collaborative edition, compilation and execution of programs in a synchronous distributed fashion, and includes advanced tools for communication, coordination and workspace awareness. The article analyzes some usability issues as well as limitations and weak points as the basis for developing future versions of the system.

1. Introduction

The advances in network and collaboration technologies enable the creation of powerful collaborative environments, of which many activities and disciplines can take advantage. Thus, Programming, which is a complex and creative task, can be supported and enhanced using groupware systems and distributed architectures.

According to Johnson [10], the process of analysing and criticising software artefacts produced by other people is a powerful method for learning programming languages, design techniques and application domains. The increasing complexity of software projects and the more frequent distribution of the members of the development teams lead to increasing problems and difficulties in the programming phase. Collaborative Programming (CP) is illuminating these scenarios by allowing distributed programmers to work jointly on the same program or application. Previous studies [19, 15] indicate that CP does not only accelerate the problem solving processes, but it also improves the quality of the programs.

Many CP systems have been developed up to the present. We propose the system COLLECE for use in software factories and, above all, in teaching-learning processes in educational centres. COLLECE supports synchronous distributed collaboration and provides shared workspaces for the tasks of edition, compilation and execution of programs. This article describes this system. Its main contribution consists of the use of proposal-based tools to coordinate the carrying out of shared tasks, the inclusion of a structured chat for communication during collaborative programming and, lastly, the availability of a wide support for synchronous work awareness.

The first part of the article reviews some systems that support distributed or collaborative programming. The COLLECE system is presented in Section 3 and a study of the utilization of COLLECE in collaborative programming activities is described in Section 4. Section 5 looks at some usability problems; and, finally, Section 6 draws some conclusions.

2. Related systems

A number of studies have approached the problems of supporting collaborative programming on the Internet. RECIPE (REal-time Collaborative Interactive Programming Environment) [16] allows geographically distributed programmers to participate concurrently in the design, coding, testing, debugging and documentation of a program. To do this, RECIPE allows the easy conversion of single-user compilers and debuggers in collaborative applications and the integration of existent collaborative editors into the system. However, it does not register the programmers' work in order to draw conclusions; neither does it offer

specialized tools for communication and awareness, which is important for productive collaboration. The DPE environment [9] is quite similar to COLLECE. This system supports collaborative edition, compilation and execution of programs, and incorporates text-based and audio communication. However, DPE presents limited support for task coordination and awareness.

Some attempts (e.g., [4]) have been aimed at incorporating collaborative support (forum, chat, instant messaging, collaborative editing, etc.) in Eclipse [3] through plug-ins, but these are still in their initial stages. On the other hand, some highly interactive synchronous collaborative editors offer advance tools for text edition, including syntax highlighting. One of these is Gobby [5], which includes an IRC-like chat for communication, and another is SubEthaEdit [17], with member panels, tele-cursors, group scrollbars, and chat. However, neither of these provide support for program compilation and execution.

In the scope of collaborative programming learning, we can find JeCo. JeCo [14] is an integration of two systems: Jeliot 3 and Woven Stories. The former animates the execution of Java programs. The latter is a co-authoring tool with which users create documents (programs) and connections among them. In so doing, Woven Stories supports asynchronous collaboration, and a chat incorporated in the system for students' discussion allows synchronous collaboration. The system lacks facilities for shared edition and animation.

3. The COLLECE system

COLLECE (COLLaborative Edition, Compilation and Execution of programs) overcomes some of the limitations of the systems mentioned above. COLLECE allows the users to edit a program or code fragment, to compile it and to run it collaboratively. Up to now, the languages supported are Java and C.

The system, developed using Java technology, operates on client/server architecture. The data management as well as the synchronization services for implementing the synchronous collaboration are centralized in a server, whereas the distributed clients (the users executing the system) access the system from a web page. The synchronization subsystem utilizes the Java Shared Data Toolkit (JSDT) [11].

The system is used by two different actors: teacher and student. The teacher defines the work sessions and arranges the users participating in them by using management tools. A session is defined by means of a name, a type, a file containing the formulation of the problem to be solved and a schedule in which the session has to be carried out. The problem formulation includes a textual description of the objectives, requirements and constraints to be fulfilled with the creation of a program. When the students access the system, the session management tool is opened, showing a list with the sessions available. Some of them are public and others are private. Any user can access a public session, whereas it is necessary to be a member to access private sessions.

When a session is accessed in the scheduled time, the COLLECE workspace is opened (Figure 1). In order to design it, we took the semi-structured model for synchronous collaborative problem solving proposed by Bravo et al. as a base [2]. This model proposes Scripting (Collaboration Protocols) [18] to structure the high-level tasks, Language/Action Perspective [20] to express and categorize actions for users' coordination, and Flexible Structuring [12] for communication between users.

In order to carry out the programming tasks, an explicit collaboration protocol must be followed. First, the students create a program using the collaborative editor; after which they are able to compile the program, receiving a list of compilation errors. Finally, they can execute the program provided a compiled program is available. Iterations are possible between these three tasks [1]. However, despite this script, the students are free to make their own decisions on when to edit, compile and execute, and to decide who is responsible for each task. To do so, coordination tools are available in the workspace to regulate the navigation through the collaboration protocol.

Such coordination processes are modelled with a simple protocol of actions extracted from language. In order to regulate the edition turn assignment (see Figure 1-3), we identified the acts *Request the edition turn*, *Give* and *Don't give*. With these acts, a user can request the edition turn and his/her fellow users can express his/her agreement or disagreement. When all the users in the group agree, the assignment is made. Similar acts are used for coordinating when to compile and when to execute the program (see Figure 1-4, 1-5). These coordination tools support multiple proposals, that is to say, proposals coming from more than one user and, as a result, lists are required to contain the historical proposals, enabling a user to select the proposal to which he/she wants to respond.

The communication during the tasks is materialized by means of a structured chat. This chat is called *structured* because it offers a pre-established set of communication acts, aimed at providing explicit communication acts that encourage the users' participation, by reducing the writing load and focusing

the users on the task. Apart from the so-called *structured messages*, the chat also provides the users with *free text* messages and the possibility of selecting one of the last messages sent in order to reuse it.

Besides coordination and communication support, awareness support is also available so that the users can easily perceive and gain knowledge of the interaction carried out by other people in the shared workspace [6].

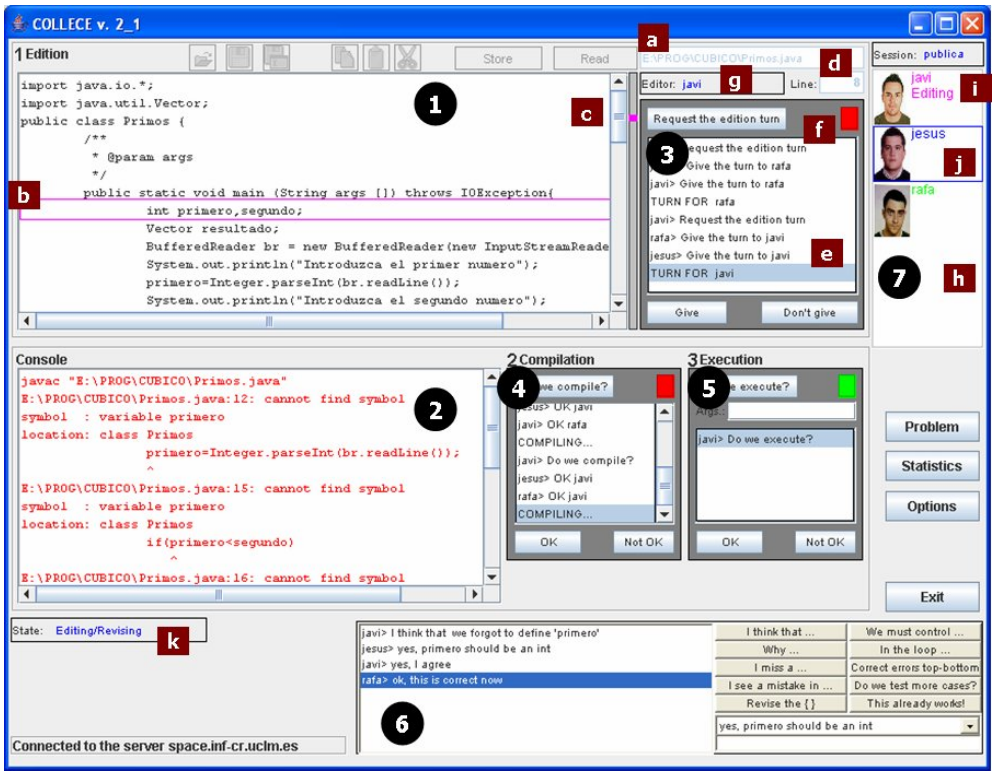


Figure 1. The main user interface of COLLECE (a snapshot of a real session).

We worked with three teachers of Programming in the participative design of the user interface. First, some paper prototypes were designed and evaluated by these teachers. A software prototype was then elaborated and put to the test in a formative evaluation. In so doing, a final design was implemented in accordance with the following main design principles:

- The three main tasks (edition, compilation and execution) are contained in a single window (the main user interface) (see Figure 1).
- The edition area consists of a shared text editor whose floor control is regulated by a turn-taking mechanism based on a coordination tool.
- The console area shows both the compilation errors and the execution outcome.
- The edition and console areas are not fully WYSIWIS (What You See Is What I See), since each user can decide which part of these areas to visualize.
- The awareness is provided mainly in the form of tele-pointers, state labels, lists of past interactions and beeps.
- The user interface also contains a structured chat, a session panel and coordination tools.

Following its implementation, a number of teachers and students used the system to carry out some collaborative programming activities with the purpose of evaluating it. Thus, some important conclusions were drawn regarding the user interface effectiveness and some refinements were made. Among them, we highlight the following improvements:

- The inclusion of a *semaphore* in the coordination tools, so that the users can perceive easily (with a green light) when and where there are proposals of other users still pending an answer.
- The addition of a visual indicator to the edition scroll bar of the remote users to indicate where the editor user is working.

– The refinement of the set of pre-defined messages included in the structured chat (see Section 4.1).

In the final COLLECE user interface four main areas are identified: the edition area at the top (Figure 1-1), the console in the middle (Figure 1-2), the chat at the bottom (Figure 1-6) and the session panel on the right (Figure 1-7). Two system functions allow the users to consult the problem formulation and the compilation statistics. The former shows a textual description of the problem to solve. The latter displays an ordered list of compilation errors and their frequency, so that the students are aware of their more frequently made mistakes.

Herbsleb and Grinter [7] see the lack of awareness as one of the major problems in distributed software development. COLLECE deals with the problem of awareness by providing a number of techniques to inform about people, their state and their actions. Specifically, COLLECE awareness is materialized by means of a number of elements: (i) session panel (Figure 1-h); (ii) global state (*editing*, *compiling* or *executing*) (Figure 1-k) and individual state (Figure 1-i); (iii) tele-pointers, in the form of a coloured rectangle drawn around the source line (Figure 1-b); (iv) lists of interactions (Figure 1-e); (v) semaphores (Figure 1-f); (vi) beeps, when actions occur; and (vii) other mechanisms (Figure 1-a, 1-c, 1-d, 1-g, 1-j).

4. The system in action

We carried out a study aimed at evaluating the system at general level and specifically the use of the chat, the utilization of the coordination tools and the quality of the awareness support. Students (N=34) enrolled in the fifth year of Computer Science studies (MSc) of Computer Science Engineering at the University of Castilla – La Mancha in Spain took part in the study. These students were organized in 17 pairs. Firstly, they received a training course in COLLECE. Then, they were asked to create a program for sorting ten previously read numbers and calculating the median. The time available for this activity was 45 minutes. The use of the system was logged for later analysis and some questionnaires were prepared to collect information from the users.

Of the 17 pairs, 14 pairs (82%) completed the activity in the time available, generating a program. We selected three variables for evaluating the programs created:

– *Well_Formed*: Quantifies to what degree a solution is well built, i.e., it does not contain compilation errors.
– *Accuracy*: Assesses whether the solution solves the problem by satisfying the requirements (sorting ten numbers).
– *Quality*: Gives an indication of the general quality of the solution.

We calculated these variables with a mixed method of quantitative and subjective analysis, and used a five-point scale ranging from 1 (very poor) to 5 (very good) to give them a value. Thus, the 14 solutions created by the students obtained an average value for *Well_Formed* of 4.71, for *Accuracy* of 3.29 and for *Quality* of 3.57.

4.1. Use of the chat

As in real life, people need to communicate when carrying out a complex task such as programming in a group. The sentence-openers approach has proved to be a potentially effective mechanism for structuring communication in a number of systems [13]. As a result, we chose to incorporate a chat with sentence openers in COLLECE. However, the study described here aims to evaluate the suitability of such a chat.

To derive a set of chat messages (sentence openers) for use in the Programming domain, we first met with some programming teachers in order to identify potentially interesting chat messages. Thus, 19 messages were initially formulated. They were filtered when considering the messages interchanged between the users during the tests of the formative evaluation (see Section 3). Finally, the teachers were interviewed again to validate a final set of 10 messages, whose texts are shown in Figure 2.

In the study, the most used message type was the free message (92%), and the least used was the structured message (3%). Analyzing the structured messages (Figure 2), "I think that…" was the most used (17 times, 38 %) and "In the loop …" was not used.

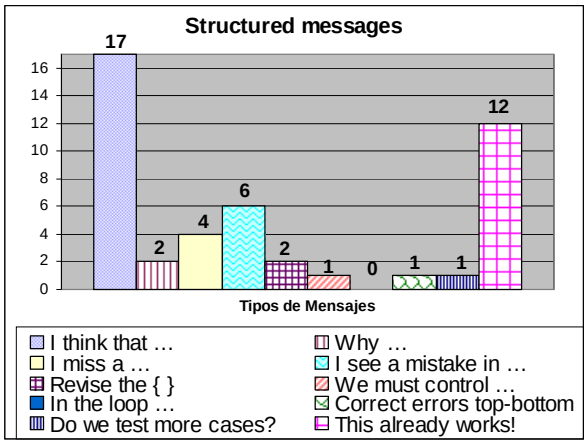


Figure 2. Use of the structured chat in the experiment.

4.2. Use of the coordination tools

Table 1 shows the number of proposals (requests of turn, compilation proposals, and execution proposals) and answers made with the coordination tools. There are two surprising results. Firstly, the number of agreements is less than expected with respect to the number of proposals. Secondly, the number of disagreements is very low. The difference between the number of proposals and the number of total responses (agreements and disagreements) means that some proposals were not answered.

Table 1. Number of coordination actions.

	Proposals	Agreements	Disagreements
Edition	205	99	3
Compilation	320	202	9
Execution	190	103	4

4.3. Students' evaluation

The students who participated in the experiment were requested to fill in a questionnaire after the working sessions. This questionnaire contained both open and closed questions to register the students' opinion about different issues of the system, such as its weak points, facility of use and of learning, and messages of the chat.

Many students identified many strong points of the system. However, our interest lies more in the weak points. At user interface level, the students indicated that there are too many areas in which to interact, and therefore that they had to pay attention to too many things at the same time. They felt the need for a better editor for editing the source code, which included the number of each line and had syntax colouring. They pointed out that the requests for edition turn, compilation and execution as well as their possible answers should be better highlighted. In regard to the structured chat, they would have liked shortcuts to select the structured messages using the keyboard, but they also suggested that structured messages are not useful since in most cases they preferred to write their own messages. The students identified some structured messages as being of little use whilst others were more interesting, something which will be considered in the future. Another issue mentioned was the CP approach: some of the students considered a more concurrent programming would be a promising advance, and others found it interesting for the users to be able to work separately, and afterwards integrate or compare their work. However, these approaches follow an instructional design different to which the system is presently based on.

Two closed questions of the questionnaire were aimed at evaluating indirectly the system usability. They were "Do you think that the user interface is easy to use?" and "Do you think that the user interface is easy to learn?". 92.68% of the students replied positively to the first question, and 97.56% thought that the user interface is easy to learn.

The students were also requested to evaluate the quality of the awareness information that some specific components of the user interface provide. Table 2 shows the averages scores once again using a five-point scale. The remote tele-pointer is the highest scoring mechanism, and the semaphore is the lowest. The session panel also scored highly, while the lists of proposals were not considered to be an efficient way of providing awareness information.

Table 2. Students' scores for the awareness tools and mechanisms of COLLECE (M: mean; SD: standard deviation).

Category	M	SD
Program with which users are working (Figure 1-a)	3.3	0.7
Remote user's tele-pointer in the edition area (1-b)	3.9	0.8
Remote user's tele-pointer in the editor's scroll bar (1-c)	3.4	0.8
Line number in which the editor user is editing (1-d)	3.5	1.1
Lists of proposals, agreements and disagreements (1-e)	3.3	1.0
Semaphore indicating proposals pending answer (1-f)	3.0	1.1
Information about who is the editor user (1-g)	3.4	0.9
Session panel (1-h)	3.8	0.7
Session panel, individual state (1-i)	3.5	0.7
Session panel, who are you? (1-j)	3.3	0.8
Global state (1-k)	3.4	1.0

5. Usability issues and lessons learned

Within the Human-Computer Interaction discipline, usability plays a crucial role in assuring the quality and success of any interactive system [8]. In collaborative systems, usability is especially important, and different evaluations are required to guarantee the best usability of a product.

Structured chat tools are frequently incorporated in synchronous collaborative systems. Despite providing some benefits, we did not find them particularly useful in our approach. The frequencies of both structured and reused messages were too low. This may have been due to us having included in the chat the possibility of writing free text messages, which were preferred by the users. Further studies are required to analyze to what extent it would be advisable to have a chat with only

structured messages, thus making the users focus on the programming task and advising them with the information contained in specific messages. Also the inclusion of an audio (and even video) channel should be further investigated.

We found significant data about the coordination tools that shows that the students did not use them properly. The low number of disagreements suggests considering the use of other coordination models that do not require two possible answers to a proposal (OK / Not OK), such as a request-release model that assigns resources when requested if they are not in use, and that are released by the owner user when he/she wants. In the specific case of the edition turn, an interesting possibility in teaching-learning settings is the mediation of the system to assign the turn.

In collaborative systems, awareness support is a component of the system's usability. Usability is related to awareness, and our hypothesis is that by improving awareness support the usability is increased. From the beginning we recognized the importance of awareness in the design of COLLECE, and thus we incorporated in the system a significant number of mechanisms. However, the students participating in the experiments were more critical as reflected in the questionnaires. In an informal and subjective way, they said that the system was easy to use and to learn, but during the experiments they had problems in using specific components such as the coordination tools. We can conclude that the system has in theory good awareness tools, but in practice they do not contribute to generate awareness information of quality, at least according to the students' opinion.

6. Conclusions

In this article we have presented COLLECE, a synchronous distributed collaborative system for supporting programming tasks (edit, compile and run programs). The system is primarily aimed at supporting collaborative learning of programming. In order to recreate classical face-to-face pair (or group) programming in a distributed configuration and, in so doing, obtain its benefits, the programming tasks are complemented with communication, coordination and awareness tools. The different designs and solutions adopted for implementing the synchronous collaboration support of COLLECE represent a proposal for synchronous groupware user interfaces.

COLLECE has been put to the test in different kinds of evaluations and studies. Good and not so good results have been obtained from them. Some specific tools have not provided the expected benefits, such as the coordination tools, the communication tool (structured chat) and the awareness mechanisms. For instance, the coordination tools require more training in the users, or even a new coordination model or a new user interface to implement them.

The aforementioned usability problems and lessons learned are a starting point for the new versions of the system. However, empirical studies to give a more rigorous estimation of both the quality of the awareness information and the usability of the system are required. Along this line, COLLECE is presently being used by university computer science students enrolled in the subject matters of Programming Fundamentals and Data Structures, so that a significant amount of evaluation data will be available. Accordingly, further work on usability and awareness problems in COLLECE will follow. For instance, radar views to improve perception and understanding of the users' work (programming tasks) need to be explored. The complexity of the system's user interface and the availability of better edition tools are aspects which will also be taken into consideration for future versions.

Finally, it is necessary to reflect on the use of the COLLECE system for professional purposes. We believe that its general approach is suitable for industry since it can support extreme programming and particularly distributed pair programming. For instance, COLLECE could be used in real projects for shared error correction, discussion of solutions or explanation of programs. Such a system would have an impact not only in the immediate productivity or quality of programming but also in the control and analysis of the process. The structuring incorporated in the system opens the door to easy analysis of the behaviour of the programmers in order to understand it and improve the programming practices (e.g., error correction behaviours, programming best practices) within a development team in future projects.

However, at present the system still has to overcome some shortcomings as regards its use in the industry. Synchronous work from different locations is allowed, but this could be a limitation when it is not possible for all the participants to work at the same time due to their different working conditions because of their different countries/regions, timetables, etc. As far as the user interface is concerned, a more advanced collaborative editor and better mechanisms for coordination, communication and awareness would be required in order for the system to provide the necessary productivity for professional contexts. From a programming point of view, the system would be enriched by supporting a greater number of

programming languages besides Java or C, and by facilitating the management of more complex programming projects made up of many programming objects of multiple types (e.g., source, object or documentation files). Regarding the whole development life-cycle, COLLECE should be easily integrated with other CASE (Computer-Aided Software Engineering) tools supporting other development tasks such as design or testing.

Acknowledgements

This research is supported by the Comunidad Autónoma de Castilla-La Mancha (Spain) in the PCI-05-006 and PAC07-0020-5702 projects, and by the Ministerio de Educación y Ciencia (Spain) in the TIN2005-08945-C06-04 project. The authors would like to thank the teachers and students from University of Castilla – La Mancha who participated in the experiments and evaluation activities.

References

[1] Bravo, C., Redondo, M.A., Mendes, A.J., Ortega, M.: Group Learning of Programming by means of Real-Time Distributed Collaboration Techniques. HCI related papers of Interacción 2004. Springer-Verlag (2004) 289-302

[2] Bravo, C., Redondo, M.A., Ortega, M., Verdejo, M.F.: Collaborative environments for the learning of design: A model and a case study in Domotics. Computers and Education 46 (2) (2006) 152-173

[3] Eclipse. http://www.eclipse.org/

[4] GILD: Groupware enabled Integrated Learning and Development. http://gild.cs.uvic.ca/

[5] Gobby: A collaborative text editor. http://darcs.0x539.de/trac/obby/cgi-bin/trac.cgi

[6] Gutwin, C., Greenberg, S.: Workspace Awareness. Position paper for the ACO CHI'97 Workshop on Awareness in Collaborative Systems. Georgia, USA (1997)

[7] Herbsleb, J., Grinter, R.: Architectures, coordination, and distance: Conway's law and beyond. IEEE Software 16 (5) (1999) 63-70

[8] ISO/IEC 9126-1. Software engineering - Product quality - Part 1: Quality model (2001)

[9] Jo, C.H., Arnold, A.J.: A portable and Collaborative Distributed Programming Environment. International Conference on Software Engineering. Las Vegas, Nevada, USA (2003) 198-203

[10] Johnson, P.M.: Reengineering Inspection: The Future of Formal Technical Review. Communications of the ACM 41 (1998) 49-52

[11] JSDT: Shared Data Toolkit for Java Technology. https://jsdt.dev.java.net/

[12] Lund, K., Baker, M.J., Baron, M.: Modelling dialogue and beliefs as a basis for generating guidance in a CSCL environment. Proceedings of the International Conference on Intelligent Tutoring Systems. Montreal, Canada (1996) 206-214

[13] McManus, M:, Aiken, R.: Monitoring computer-based problem solving. Journal of Artificial Intelligence in Education 6 (4) (1995) 307-336

[14] Moreno, A., Myller, N., Sutinen, E.: JeCo, a Collaborative Learning Tool for Programming. Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04). Washington DC, USA (2004) 261-263

[15] Nosek, J.T.: The Case for Collaborative Programming. Communications of the ACM 41(3) (1998) 105-108

[16] Shen, H., Sun, C.: RECIPE: a prototype for Internet-based real-time collaborative programming. Proceedings of the 2nd Annual International Workshop on Collaborative Editing Systems. Philadelphia, Pennsylvania, USA (2000)

[17] SubEthaEdit: Collaborative text editing. http://www.codingmonkeys.de/subethaedit/

[18] Wessner, M., Hans-Rüdiger, P., Miao, Y.: Using Learning Protocols to Structure Computer-Supported Cooperative Learning. Proceedings of World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA'99). Seattle, Washington, USA (1999) 471-476

[19] Williams, L.A., Kessler, R.R.: All I really need to know about pair programming learned in kindergarten. Communications of the ACM 43(5) (2000) 108-114

[20] Winograd, T.: A Language/Action Perspective on the Design of Cooperative Work. CSCW: A Book of Readings. Morgan-Kaufmann (1988)