The 3rd International Workshop on Global Software Development

Co-located with ICSE 2004
International Conference on Software Engineering
Edinburgh, Scotland
May 24, 2004

Table of Contents

Workshop Introduction 1

Session 1: Feasibility of Global Software Development

Transitioning from a Co-Located to a Globally-Distributed Software Development Team : A Case Study at Analog Devices Inc. 4
David Boland, Analog Devices;
Brian Fitzgerald, University of Limerick

Global Software Development Process Research at Siemens 8
Matthew Bass and Daniel Paulish, Siemens Corporate Research

Efficient Maintenance Support in Offshore Software Development: a Case Study on a Global E-Commerce Project 12
Zheng Yan, Helsinki University of Technology

Risk Management in Global Software Development: A Position Paper 18
Rafael Prikladnicki and Marcelo Hideki Yamaguti,
Pontifícia Universidade Católica do Rio Grande do Sul

Session 2: Strategies for Success in GSD

Can Global Software Teams Learn From Military Teamwork Models? 21
Elizabeth Hargreaves and Daniela Damian, University of Victoria

Designing the Inter-Organizational Software Engineering Cooperation: An Experience Report 24
Hans Nissen, RWTH Aachen, Informatik 5

Towards a Model of Awareness Support of Software Development in GSD 28
James Chisan and Daniela Damian, University of Victoria

Peer-to-Peer Remote Conferencing 34
Fabio Calefato, Filippo Lanubile and Teresa Mallardo, University of Bari

Test-Driven Global Software Development 38
Bikram Sengupta, Vibha Sinha and Satish Chandra,
IBM India Research Laboratory;
Sharath Sampath and K. Guru Prasad, IBM Global Services India

Session 3: Research Methodologies and Challenges in GSD

Using Iterative and Incremental Processes in Global Software Development 42
Maria Paasivaara and Casper Lassenius, Helsinki University of Technology

Communication Problems in Global Software Development: Spotlight
on a New Field of Investigation... 48
Sébastien Cherry and Pierre Robillard, École Polytechnique de Montréal

An empirical study on Global Software Development: Offshore
Insourcing of IT Projects 53
Rafael Prikladnicki and Jorge Audy, PUCRS;
Roberto Evaristo, University of Illinois

Analyzing Intercultural Factors Affecting Global Software
Development – A Position Paper.. 59
Philippe Kruchten, University of British Columbia

The Benefits and Limitations of Knowledge Management in Global
Software Development.. 63
Torgeir Dingsøyr, SINTEF ICT;
Knut-Helge Rolland and M. Letizia Jaccheri, NTNU

The 3rd International Workshop on Global Software Development May 24, 2004

http://gsd2004.uvic.ca

Daniela Damian
University of Victoria, BC, Canada
danielad@cs.uvic.ca

Filippo Lanubile
University of Bari, Italy
lanubile@di.uniba.it

Elizabeth Hargreaves and James Chisan
University of Victoria, BC, Canada
{elizabeth.hargreaves, chisan}@cs.uvic.ca

The goal of this one-day workshop is to bring together participants from the research community as well as software industry in order to explore both the state-of-the-art and the state-of-the-practice in global software development (GSD). Increased globalization of software development creates software engineering challenges due to the impact of temporal, geographical and cultural differences, and requires the development of methods and technologies to address these issues. This workshop is being organized to foster interaction between practitioners and researchers in order to address the pressing issues in this area. Practitioners experiencing challenges in GSD will share their concerns and successful solutions and learn from research about current investigations. Researchers addressing GSD will gain a better understanding of the key issues facing practitioners and share their work in progress with others in the field.

This third workshop edition features an innovative program that discusses the feasibility of GSD, successful strategies for GSD, as well as methodologies and challenges in conducting research in this growing area of interest.

1. Workshop organization and program

It has been our tradition in the last three years at this workshop to provide practitioners and researchers with an opportunity to explore current challenges within the growing field of global software development (GSD). This year we solicited papers in the following categories:

(1) Case studies of GSD,
(2) Theories of communication, coordination, collaboration and knowledge management in GSD,
(3) Methods and tools to address challenges of GSD,
(4) Empirical evaluations of effectiveness of global software projects and
(5) SE methodologies & processes for GSD.

In response, we received 17 submissions (11 technical papers and 6 position papers), out of which 14 were accepted for presentation and publication in the workshop proceedings. These papers are available at:
http://gsd2004.uvic.ca/upload/ContentTable.html

To our delight, the demographics of the submissions addressing issues of GSD are truly international this year. Authors are joining the workshop from North America (Canada, US), South America (Brazil), Europe (Italy, Ireland, Finland, Norway and Germany) and Asia (India and Singapore). Research results are being reported from case studies at large multi-national corporations such as IBM India, Nokia, Siemens and Analog Devices, whose industrial sites span the globe, including: Finland, US, Ireland, Singapore, India, and Brazil.

Based on these submissions, we decided to organize the workshop into four sessions, each session featuring a different topic of discussion. The first session opens with a keynote talk by Philippe Kruchten from UBC, Canada, who will discuss the challenge of cultural differences in GSD. This is followed by three sessions in which papers will be presented in the following categories: (1) The Feasibility of Existing GSD Practices, (2) Successful Strategies for GSD, and (3) Research Methods and Challenges.

During each session authors will briefly summarize their work and form a panel to address the session topic. Each session will be facilitated by a session chair who will mediate discussions of this topic with the panel members and the entire workshop audience.

The topics of the sessions in our program, together with the relevant papers, are described below. This is followed by a section summarizing background information that motivates research in the area of global software development.

1.1 Feasibility of Global Software Development

The long-term feasibility of global software development (GSD) as standard practice remains undetermined due to its relative novelty. This session will address some of the existing concerns and explore feasibility issues in depth. In particular, we will attempt to determine the appropriate use of GSD practices in relation to software engineering projects as a whole as well as during specific project phases. Intercultural, logistical, technical and fiscal limitations will be considered. Four papers that relate directly to feasibility concerns will be presented and discussed. Boland and Fitzgerald's paper will allow workshop participants to analyze transitioning from co-located to globally distributed environments within the framework of a specific case study. *Bass and Paulish*'s contribution will provide insight into the GSD strategies used by Siemens—one of the most globally distributed software companies in the world. Yan's case study will highlight the GSD challenges experienced by Nokia during the maintenance phase of an e-commerce project. Finally, Prikladnicki and Yamaguti's position paper will highlight risk management issues related specifically to GSD projects. The intention of this session is to foster discussion, isolate key issues for further research and establish strategies for future collaboration.

1.2 Strategies for success in GSD

While feasibility may be questioned by some, others may argue that globalization is by now well-established, and to suggest that software development should be immune is foolish. Already, the trend toward outsourcing is underway, especially in software development and support. The question is not whether GSD is possible, but how to improve it and how to make it more efficient. A series of options presented in five papers will be considered: from tool support to revising the development process to engendering new attitudes among software developers. The session will begin with Hargreaves and Damian's position paper describing how military-styled team dynamics might be used as an example of how to build highly successful GSD teams. Nissen follows, describing how cooperation models were used as a basis to manage expectations during a successful GSD project. Chisan and Damian contribute a model for how awareness of development artifacts can improve cooperation and communication among software developers. Calefato et al. take a tool-based approach, by describing in detail a peer-to-peer conferencing tool that could address communication needs in GSD. Finally, Sengupta et al. posit a process approach of test-based-programming to maintain a consistent project-wide view of requirements. The goal of this session is to discuss and disseminate practices, strategies and research endeavors among participants to promote research and the state of the practice.

1.3 Research methodologies and challenges in GSD

The need for empirical research in GSD is indisputable. Whether the research is about success factors of global teams, the development of theories about GSD or evaluating proposed strategies, the role of empirical data from real-life software industrial settings is critical. The fundamental question that emerges is which research methodologies, strategies and techniques are appropriate for the collection and analysis of empirical data such that we achieve a systematic advancement of knowledge in this growing area of research.

Paasivaara and Lassenius, and Prikladnicki et al present case studies in which interviews and project documents were used to identify collaboration practices in GSD. In contrast, Cherry and Robillard study ad-hoc communication through observations of global teams. Further, Kruchten proposes the study of intercultural factors through a combination of ethnographic studies, content analyses, surveys and experiments. This strategy is similar to that of Dingsoyr and colleagues who plan to implement a multidisciplinary approach in studying global teams in both commercial and open source projects. This session will attempt to identify methodological issues such as: what are the challenges in applying "traditional" empirical methods in the context of geographical distribution of study participants, and whether factors such as trust and cultural differences that affect global software development are having an impact on how research is carried out in this field.

2. Global software development background and motivation for research

Global software development (GSD) is increasingly becoming common practice in the software industry. The ability to develop software at remote sites allows organizations to ignore geographical distance and benefit from access to a larger, qualified resource pool with the promise of reduced development costs. However, the increased globalization of software development creates software engineering challenges due to the impact of temporal, geographical and cultural differences, and requires development of methodologies, techniques and technologies to address these issues.

This is the third Global Software Development workshop organized at the International Conference on Software Engineering (previous two workshops [3][4]), and it is a continuation of the ICSE workshops on Software Engineering held over the Internet from 1998 to 2001. The issues discussed at the workshop have focused on addressing problems due to cultural and organizational differences in multi-site and multi-national software enterprises as well as the technical challenges experienced by software development at a distance. This is emphasized in the workshop reports [1][2][4] which discuss the challenges of engineering software in geographically distributed settings, and indicate that further research needs to address technical and social issues in global software development.

Global software development has been and continues to be a phenomenon fueled by factors such as access to a large and specialized labor pool, reduction in development costs, global presence and proximity to the customers. While we are witnessing reports of successful global teams, research reveals that distance contributes to heightened complexity in organizational processes. Primarily, processes of communication, coordination and control are affected by distance, with direct consequences on how software is defined, constructed, tested and delivered to customers, as well as how its development is managed. Furthermore, inherent cultural issues are perhaps the most confusing, yet intriguing aspect of global teams. Stakeholders who are expected to work together as a team do so despite their diverse attitudes towards hierarchy, time management and adversity to risk.

These are only some of the factors that bring challenges to managing software projects developed in geographically distributed structures. Understanding the intricacies of this complex phenomenon is critical in framing research directions that aim at improving global software development practice. There is a need for tools and techniques that not only improve development processes but also address organizational and social issues in global software development. The previous workshops each represent one more step in identifying and understanding issues in the complex phenomenon of global software development. In particular, the empirical evidence and discussions during the workshop in the last years indicate that technology is only a small part of enabling effective global teams; there is a strong need to address the study and practice of global software development from a multidisciplinary perspective, in which issues of social nature are as important as those of technical nature.

In this workshop we intend to continue fostering fruitful interactions between industry practitioners and researchers, and help establish a community of interest in this area. Industry practitioners experiencing challenges in GSD will be encouraged to share their own solutions and learn from research about current investigations in this area. Researchers addressing GSD will have the opportunity to gain a better understanding of the key issues facing industry practitioners and share their work in progress with others in the field.

4. Workshop main organizers' background

Daniela Damian is an Assistant Professor at the University of Victoria, BC, Canada, where she holds the NSERC University Faculty Award. Daniela is the director of the SEGAL Labs (Software Engineering Global interAction Laboratories) at University of Victoria, and was the primary contact and co-organizer of the ICSE Workshops on Global Software Development 2002 and 2003. She has been acting on the Program Committee of conferences and workshops in the areas of requirements engineering and distributed software engineering. She is an editorial board member of the Journal of Requirements Engineering, Associate Editor for the Int'l Journal of Human-Computer Studies. Currently she is the Guest Editor of the special journal issue on Global Software Development in the Journal of Software Process: Improvement and Practice set to appear in early 2004.

Filippo Lanubile is an Associate Professor at the University of Bari, Italy. While at University of Maryland (1995-1997) he was a recipient of the NASA Group Achievement Award (1996). In 2003 he was the co-organizer of the Workshop on Global Software Development and acted as a member of the program committees for the Int'l. Conf. on Empirical Software Engineering, the Int'l Symposium on Software Metrics, and the Workshop on Cooperative Support for Distributed Software Engineering Processes. Currently he is the Program Co-Chair of the Int'l Workshop on Program Comprehension 2004 and Program Co-Chair of the Int'l Symposium on Software Metrics 2005.

Acknowledgements

We would like to thank all of the workshop authors and participants for contributing to the workshop this year.

References

[1]. Damian, D., "Workshop on Global Software Development", SIGSOFT Software. Eng. Notes, vol. 27, no. 5, September 2002

[2]. Lanubile, F., Damian, D., Oppenheimer, H. "Global Software Development: Technical, Organizational, and Social Challenges", SIGSOFT Software. Eng. Notes, vol. 28, no. 6, November 2003

[3]. International Workshop on Global Software Development 2002, http://www.cis.ohio-state.edu/~nsridhar/ICSE02/GSD

[4]. International Workshop on Global Software Development 2003. http://gsd2003.cs.uvic.ca

Transitioning from a Co-Located to a Globally-Distributed Software Development Team : A Case Study at Analog Devices Inc.

David Boland
Analog Devices
david.boland@analog.com

Brian Fitzgerald
University of Limerick
bf@ul.ie

Abstract

Global software development has become an extremely important issue for organizations at present in the climate of increasing tendency towards globalization and global outsourcing. A number of studies have been conducted which have identified a set of problematic areas which are common across projects, including language and cultural differences, trust factors, communication across temporal and spatial distances, lack of shared contextual awareness. This study of global software development at Analog Devices Inc. (ADI) is especially noteworthy for a number of reasons. Firstly, the project has recently moved from a co-located to a globally-distributed one, and thus the team had already had experience of being co-located, a factor that has not typically been the case in the studies published to date where teams are being established who have not previously been co-located. Also, as language and cultural factors were not an issue, the study was able to focus on the problems of communication over temporal and spatial distances. The study discusses how ADI attempted to address these problems and identifies the initiatives that worked well, and, more importantly, those that did not work as well. Among the findings was the fact that trust, which had been very solidly established among team members during co-location, was significantly eroded as the project team was reconstituted on a distributed basis.

1. Introduction

There have been several documented studies on globally distributed software development teams [e.g. 1,2,3,5,7, 9,10]. A common feature in most of these studies, however, has been that the teams at the various development sites have had little or no previous experience with each other. Also, many of the case studies have involved very large development teams and substantial geographical and temporal distances (i.e. greater than 8 hours). This particular case study, however, was able to observe a very small development team (less than 20 developers), that had worked together for four years and were being redistributed into a global development team across two development sites; one in the United States and the other in Ireland. Many common global development problems including language and culture were not an issue and this allowed us to concentrate on how communication and temporal problems affected the group and how they attempted to overcome them.

The paper is structured as follows: The next section provides some background on the case study company, Analog Devices Inc. Following this, the procedures and processes that were established in the move from co-location to a distributed team are identified. The next section discusses the success of these procedures and processes, and also identified the problematic areas where these did not work as well. Finally, the conclusions and implications of the study are addressed.

2. The Company

Analog Devices Inc. (ADI) is a world-leading semiconductor company specializing in high-performance analog, mixed-signal and digital signal processing (DSP) integrated circuits (ICs). ADI currently has a worldwide workforce of approximately 8,600 employees, including 3,100 engineers. There are development/manufacturing facilities in the United States, Ireland, United Kingdom and the Philippines.

Analog Devices is one of the few semiconductor companies that have an internal division that provides automatic test equipment (ATE) for the ICs the company produces. Analog Devices' ATE division is called the Component Test Systems (CTS) division. The latest ATE platform at CTS has been in development since 1999 and for all that time the entire development team, both hardware and software engineers, have been co-located.

In 2003, it was decided to distribute some of the team members to the development facility at Limerick, Ireland. The primary purpose for the relocation was to ensure that CTS was better represented at the remote site. This would provide better support to the local customers and their concerns/issues would be more accurately relayed to CTS.

3. Creating a Globally Distributed Development Team

There are many problems to be addressed when establishing a globally distributed development team, including, for example, language and cultural differences, trust factors, communication across temporal and spatial distances, lack of shared contextual awareness [2, 4, 6, 8, 9]. CTS, however, believed that the creation of their team would be successful as some of these problems would not be an issue. The problems included:

1. **Language**. All the members of the team spoke English and used a common vocabulary for identifying specific hardware or software components. Therefore, the team should have no difficulty understanding each other.
2. **Culture**. Although not all members of the team were from the same geographical region, they had been working together for four years at the time of the move to a distributed team, and thus had developed their own 'CTS' culture. Unintentional rudeness, hostility or other communications issues should not be a problem.
3. **Trust**. The developers had established strong levels of trust between each other as a result of working together for a long time.

Therefore, CTS was able to concentrate on addressing the remaining global development problems of communication across temporal and spatial distance, and shared contextual awareness. The following procedures/processes were enacted to address these issues.

Single Software Manager

Due to the size of the development team, it was decided to continue with one software manager for all developers across all sites. The software manager is responsible for assigning tasks that will reduce cross-site dependencies especially with regard to expert dependencies (i.e. assign tasks to the particular subsystem expert directly or have experts and developers co-located).

Weekly Task Report

To facilitate the work of the software manager each developer was required to submit a task report at the beginning of each week. The report includes a list of their specific goals for the week and a summary of their progress for the previous week. The report also indicates if the developer intends to make any deliveries during the week (i.e. check their work into the main source tree). This reporting process enables the software manager to be aware of work progressing across all the development sites and provides the necessary information to coordinate tasks among the developers.

Delivery Report

A new check-in procedure was introduced to ensure each developer was kept aware of all the work progressing at each development site. At check-in the developer must submit a report outlining a description of the changes/features they are checking into the main source tree. This description includes the specific files (source code, documentation, etc) that have been changed or added. The report also includes the primary purpose behind the delivery and how to test the changes/new features.

New Communication Tools

CTS developers rely heavily on informal communication to design, implement and debug their systems. To help facilitate informal communication across the development sites, developers were encouraged to use AOL's Instant Messenger (IM). Microsoft's Net Meeting was also made available to all developers.

Quarterly meetings

Once a quarter all the developers are gathered together to meet face-to-face for one week. This business trip is called a 'sync up' trip. Development goals and future projects are discussed but the primary purpose for the trip is to increase the team's morale and to maintain the camaraderie between the developers.

4. Results

The globally-distributed development team has been operational for four months. In general, the group is performing well but communication and temporal problems have resulted in reduced productivity, trust and morale levels. The following are the procedures and processes which have been initiated and seem to be working well:

Software manager and weekly task reports – Reduced inter-site dependencies

The software manager was able to make good use of the weekly task reports and has been successful at assigning the majority of tasks between the sites appropriately.

Delivery reports - Maintained awareness and trust levels

The delivery report has been successful at maintaining group awareness and has made it easier for each developer to know who is working on what, who are the experts on particular subsystems, the problems being addressed and the problems outstanding. This procedure, if combined with the absence of other communication problems, was perceived to be sufficient at maintaining trust levels between the developers. Other communication problems, however, did become evident and thus eroded this procedure's effectiveness in this area.

Quarterly sync-up meetings – Maintained morale and motivation levels

These trips have proven to be very successful and developers have commented on feeling 'energized' and highly motivated after meeting with all the team members.

Friendship – An important contributor to awareness

Some of the developers had become good friends during the period they were co-located. These friendships proved invaluable to maintaining informal communication channels between the development sites. When these developers needed to discuss an issue, through either synchronous or asynchronous communication channels, they invariably discussed other, unrelated, issues. Discussions of this nature are a critical component of software development [10]. At CTS these discussions gave each developer greater insight into the particular operations at each site and resulted in greater overall awareness.

However, it was also the case that in some areas, the procedures and processes initiated did not work as well as anticipated:

Communication Tools - Not as effective as hoped

All the developers took the opportunity to use IM but they found that the tool was only adequate for transmitting yes or no style questions. Net Meeting was never used by the developers due to the effort it required to setup and use. Both, IM and Net Meeting are primarily synchronous communication tools and developers indicated that they prefer to use the telephone to converse if an opportunity for synchronous communication is available. This suggests that some of the technology for synchronous communication which is commonly provided does not afford developers sufficient richness as a communication to be perceived as useful.

Communication levels - Did not match co-located levels

Overall, the communication bandwidth was not adequate to compensate for the richness of informal communication between co-located developers. As a consequence, minor issues, usually discussed through informal communication channels [10], were not discussed between the development sites. This resulted in the introduction of bugs into the system. Also feedback on successful deliveries, an important contributor to morale, was completely lacking. Feedback on successful deliveries had in the past been usually done at CTS through informal channels via chance meetings with end users or other developers. Due to the lack of informal communication, however, many developers have stopped getting this feedback and thus their morale has been adversely affected.

Remote experts - Led to productivity and trust problems

When a developer is working with unfamiliar code and the subsystem expert is co-located, the developer would seek their advice on the change/feature they intended to make. The time taken to converse with the expert is usually only a few minutes but if the expert is remote this time can become several hours or even days. Thus in the interests of rapid development most minor changes are made to the subsystem without consulting the expert [10]. These changes, however, may have overlooked subtle design considerations within the subsystem and thus have introduced bugs or other problems.

When these problems become evident (either through expert analysis of the delivery report or errant runtime behavior) significant time is wasted at each development site to address the issue. The level of trust between the expert and the particular developer is also reduced. Merely raising the newly discovered problem with the group can also adversely impact morale, especially if the expert is not very 'diplomatic' at pointing out the problem. Thus, the previous high level of morale and trust that had been built up over the years between developers was possibly eroded somewhat.

Time zone differences - Led to productivity loss

Time zone differences are fundamental source of difficulties for a globally distributed development team [1, 6, 7, 10] and this was no different at CTS. Each day developers arrive to work with an inbox full of questions and other issues from the remote site. To resolve these issues takes significant time for the developer and thus their productivity is affected. Developers indicated that the majority of these issues could actually be resolved quickly, if synchronous communication was available. Also, even when synchronous communication would be possible, the extra effort to try accomplish a rich and detailed interaction through a narrow communication channel such as IM would also affect productivity.

5. Conclusion

The new global development team at CTS is performing at acceptable levels. It is interesting, however, that given the ability to concentrate on communication and temporal problems the team could not retain the level of productivity it enjoyed when all the members were co-located. Most of the loss in productivity was a result of inadequate processes that were established to address the geographical and temporal distances. There were also several unexpected problems, including the effort required to maintain a globally-distributed development team. This has resulted in an increased workload for some of the developers and thus resulted in a drain on their productivity.

Today, some developers are occasionally failing to follow all of the processes due to project deadlines, workload or other issues. Thus productivity, awareness, trust and other areas will begin to be adversely impacted unless the process can be improved.

Clearly a zero cost, synchronous communication channel that can work around time zones would drastically improve GSD – so we need either a Star Trek transportation device or a time machine! In the somewhat unlikely event of either appearing in the foreseeable future, we will continue to work on the problems and issues identified here.

6. References

[1] Kiel, L., "Experiences in Distributed Development: A Case Study", *ICSE Workshop on Global Software Development*, May 2003.

[2] Pyysiainen, J., "Building Trust in Global Inter-Organizational Software Development Projects: Problems and Practices", *ICSE Workshop on Global Software Development*, May 2003.

[3] Passivaara, M., "Communication Needs, Practices and Supporting Structures in Global Inter-Organization Software Development Projects", *ICSE Workshop on Global Software Development*, May 2003.

[4] Damian, D., Chisan, P.A., and Corrie B., "Awareness meets requirements management: awareness needs in global software development", *ICSE Workshop on Global Software Development*, May 2003.

[5] Oppenheimer. H..L., "Project Management Issues in Globally Distributed Development", *ICSE Workshop on Global Software Development*, May 2002.

[6] Carmel, E., and Agarwal, R., "Tactical Approaches for Alleviating Distance in Global Software Development", *IEEE Software*, March/April 2001.

[7] Battin, R.D., Crocker, R., Kreidler, J., and Subramanian, K., "Leveraging Resources in Global Software Development", *IEEE Software*, March/April 2001.

[8] Herbsleb, J., and Mockus, A., "Challenges of Global Software Development", *7th IEEE International Software Metrics Symposium*, April 2001.

[9] Herbsleb, J., Mockus, A., Finhost, T.A., and Grinter, R.E., "Distances, Dependencies, and Delay in a Global Collaboration", *ACM Conference on Computer-Supported Cooperative Work*, December 2000.

[10] Herbsleb, J., and Grinter, R.E., "Splitting the Organization and Integrating the Code: Conway's Law Revisited", *ICSE Workshop on Global Software Development*, May 1999.

Global Software Development Process Research at Siemens

Matthew Bass, Daniel Paulish
Siemens Corporate Research, Inc.
Princeton, NJ 08540
{ Matthew.Bass, Daniel.Paulish } @Siemens.com

Abstract

Siemens Corporate Research (SCR) is the research and development unit of Siemens USA. The Software Engineering department of Siemens Corporate Research spends much of its time doing consulting for Siemens Business Units. As a result, we have been involved in a large number of software development projects varying in size, complexity, and domain. Many of these projects were developed with globally distributed teams. Over the years, we have identified best practices, and begun to organize these practices into more cohesive set of development processes focused on issues related to global development. This paper describes our experience with experimentation, lessons learned from one specific project, and suggests future steps for global software development (GSD) within Siemens.

1. Introduction

Siemens is one of the largest developers of software intensive systems in the world. With a presence in over 190 countries, it is also one of the most globally distributed. As software products are growing in complexity and the organizations that develop them are also growing in staff size, Siemens business managers are seeking new approaches to get new software products quicker to market, while reducing their overall development investments. One of the strategies that Siemens has adopted is to move some of its software development to low cost countries. The implications of such a decision are not entirely known. The associated risks, required changes in the development process, needed infrastructure changes, and required modifications to the management practices for successful global development are not fully known.

2. Data Processing System 2000

The Data Processing System 2000 (DSP2000) is a software system for acquiring and processing meter data, from electrical, gas, and water meters. The meter data is stored and processed so that billing determinants can be calculated for periodic transfer to a billing system. The billing system generates the bills for energy or resource consumers.

The development for DSP2000 was done at four sites in three countries. SCR staff acted as project manager and lead architect, and developed one component for this project. The product is currently being successfully sold and distributed. This section describes our experience with the DSP2000 project as it relates to GSD, section 3 then highlights some of the lessons learned from our experiences with GSD projects, and section 4 describes planned next steps towards improving the state of the practice of GSD within Siemens.

2.1 Global Analysis

Global Analysis (GA) [1][2] is a technique for analyzing, categorizing and documenting factors that influence the architecture and project management of a system. In the DSP2000, GA was completed early on during the high-level design for the DSP2000 project. Three types of factors were considered; organizational factors, technological factors, and product related factors.

2.1.1 Organizational Factors

Organizational factors may apply only to the project at hand (as in the case of schedule and budget), or can impact every product developed by that organization (as in the case of culture, development site(s) location, and software development process).

Two examples of organizational influencing factors in the DSP2000 project were:

- Technical skills were in short supply, prior products were Unix-based with local user interfaces, and marketing required new products to be Windows-based with web-based user interfaces.
- Time to market was critical. The market was rapidly changing, and it was viewed as critical to quickly get some limited features of the product to potential users so their feedback could be solicited.

Two strategies were adopted to address these organizational factors. In order to mitigate the lack of technical expertise, it was decided that this project would exploit expertise located at multiple development sites, and to invest in training courses early in the development. As a result of the criticality of time to market, it was decided that the product would be released incrementally. In this way, release dates could be met even if some features were missing. Additionally, a design strategy was followed to reuse the current data-acquisition system, and attempt to use third-party components wherever possible.

2.1.2 Technological Factors

Technological factors may limit design choices to the hardware, software, architecture, platform, and standards that are currently available. Technology, however, changes rapidly, and so if it is the case that the architecture has even a reasonably significant lifetime, then it should be designed with this in mind.

Two examples of technological factors that influenced this project were:

- An object broker was necessary for meeting the scalability and availability requirements within a distributed hardware configuration.
- The database system was expected to change over time. Oracle 8 was initially specified, but it was known that new versions would become available, and some customers would prefer other vendors.

Microsoft COM was selected to as the object broker, and a layer was designed in the architecture to abstract the database in anticipation of future database changes.

2.1.3 Product Factors

Product factors include features of a product as well as qualities like performance, dependability, security, and cost.

Two examples of product factors that influenced this project were:

- This product was to be designed as a product line. In order to support a product line architecture, the graphical user interface (GUI) had to be able to accommodate many different types of users for different applications.
- The required scalability and anticipated performance requirements of the system were another influencing factor. The DSP2000 was intended for industrial and commercial applications where thousands of meters would be handled. While it wasn't originally specified for the residential market, where millions of consumers would be required, it was known that this might be a future possibility.

A web-based GUI was select to address the needed flexibility. In order to allow for potential unknown market performance requirements, we anticipated that a scalable distributed platform was necessary.

2.2 Project Planning

The DSP2000 software development was planned as a sequence of incremental engineering releases, the first of which consisted of a "vertical slice" of the architecture, which functioned as a prototype of the architecture. The last planned release was the first set of functionality that was sold as a package to a customer.

We found that a six to eight week cycle time for each iteration worked best. Some of the release dates were driven by trade shows, at which time a new release with the latest functionality was required. Particularly in light of our global development, we found that one of the best means of communication was via the system itself. It was difficult to fully understand and discuss the explicit and implicit requirements without an appropriate prototype. The system itself turned into the common language for all involved, facilitated by the web-based GUI.

The planning process itself was complicated by the distributed nature of the project. What ended up working well was to distribute drafts of the proposed schedule and task assignments for each incremental release to the team members. Often, we would get feedback in the form "This feature cannot be achieved in the time frame provided", or "I am planning a vacation during these weeks". A second version of the schedule committing the release dates and feature sets would then be distributed.

Another item that is useful in global project teams is an explicit statement of the overall project goals. An example of such a statement is "Quality will have a higher priority than schedule, which will have a higher priority than functionality." Such an explicit statement helps project managers make the inevitable trade-offs that must be made right before a release. We have found in the past that cultural bias exists that will influence such trade-offs at a local level, unless such an explicit statement of goals exist.

2.3 Project Management

Each development site had a local manager to manage the team members at that site. There was also an overall project manager, and a project manager for each software application package development. As a result there was overlapping management responsibility for achieving the project goals. These managers had to negotiate individual work assignments. In practice, however, most potential conflicts were resolved when the proposed development plan was distributed for feedback.

The chief architect was responsible for decision making and resolving technical conflicts for the application package. Analogous to the overall project manager, the chief architect was the overall technical manager. In practice, both the technical manager and the project manager reviewed key technical decisions.

An engineer was assigned responsibility to each subsystem. This engineer was responsible for the detailed design and implementation of this subsystem.

Project status tracking was done during weekly teleconferences. Each team member was encouraged to report on his development progress and to raise information or issues to be shared with other team members.

3. The Influence of Global Development

While the decision to develop DSP2000 across multiple sites was primarily motivated by the lack of resources with the required technical skills, the implications of that decision were felt in the project planning, project management, architecture, and design of the system.

Communication is a key issue in most projects, but additional barriers to effective communication exist in globally distributed projects. Several strategies were found to be useful in the DSP2000 project in overcoming the communication barriers. Those strategies include:

- Explicitly documented project goals – in the absence of clear direction, local cultural and personal biases are going to influence decisions. The resulting choices may not be in line with the overall goals of the project.
- Incremental development – an incremental release schedule with fairly short cycles helps to facilitate communication, and highlight ambiguities and misunderstandings. While this can be useful in many projects, co-located teams may have options that are not available to a globally distributed team.
- Internationally aware calendar – it was important that weekly teleconferences take place to monitor status, and highlight issues. It was important (and often difficult) that time zones and local holiday schedules be taken into account when scheduling these meetings.
- Well-partitioned architecture – in order to facilitate work break down across multiple sites, the architecture needed to reflect the organizational structure of the project. There needed to be well-defined components or subsystems with understood dependencies for each site. These components or subsystems also needed to take into account the technical skills of the staff at the responsible development sites. As it turned out, the decision to distribute the development globally had a large impact on the architecture.
- Communication of progress – in the DSP2000 project, the Uniform Resource Locator (URL) for the test system was made available for all the team members and their management. This was a big morale boost for the team, since everyone was aware of the rapid progress being made. The result was a much greater sense of team then would otherwise have been possible in a globally distributed project.

4. Current Research Focus

We were pleased with our experience on the DSP2000 project. We feel that many of our approaches were validated based on the success of this project. Ideally the decision to use distributed development teams would result from influencing factors relating to the project in question. More and more, however, this is not the case. Distributing development to low cost countries has become a cost-saving strategy for many organizations. Siemens is no different. It is not clear what the impact of such an approach has on the bottom line. While the hourly development cost may be reduced, extra effort is likely to be spent on project management, architectural design, requirements engineering, and so forth.

SCR is currently in the process of codifying past experience in the form of questionnaires, checklists, processes, and other decision aids to assist in the successful application of global software development. We are attempting to correlate project characteristics with proven strategies in order to better establish criteria for success for given projects.

One area where we are planning additional work is the experimental application of a reference process for GSD. Our process includes best practices from requirements engineering, software architecture design, and organizational patterns. Engineering rules of thumb are used to plan projects, specify the size of software components, the division of responsibilities between a central product management team and remote component development teams, metrics, tools, and operational procedures. The experimental projects are used as case studies to further support the identification of best practices.

We feel that we have a good start in understanding some of the issues related to successfully managing a global software development project. We now need to further substantiate, refine, and transfer our approach to the Siemens operating companies.

References:

[1] Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture,* Addison Wesley, Massachusetts, 2000.

[2] Paulish, D.J., *Architecture-Centric Software Project Management,* Addison Wesley, Massachusetts, 2002.

Efficient Maintenance Support in Offshore Software Development: a Case Study on a Global E-Commerce Project

Zheng Yan
Helsinki University of Technology
Software Business and Engineering Institute
POB 9210, FIN-02015 HUT, FINLAND
Zheng.Yan@hut.fi

Abstract

Software maintenance is a very important phase in software development. It generally occupies the most of development life cycle in order to ensure software quality. This paper takes an e-commerce project as an example to study how to efficiently provide software maintenance support in offshore software development for a global deployed software product. Through interviews and a survey to the project developers, authors summarize the good methods and approaches used in its maintenance that greatly helped its success. Meanwhile, the authors also study lessons that influenced its efficient maintenance (e.g. extra workload caused by performance tuning, troubles due to sharp time-difference, problem-reproducing difficulty caused by testing environment difference and slow code transfer). Suggestions for further improvement are also proposed based on real experiences in order to benefit similar software development in the future.

1. Introduction

Offshore software development generally means that software is developed through collaboration of a team in an emerging country. It is one type of distributed software development that is adopted by many companies [2]. Lower cost, plentiful skilled staffs, high quality and trustworthy are main attractions for software development abroad. However, the offshore software development also faces difficulties and risks on decision-making, coordination, execution, communications and project management. There are many issues worth studying in the offshore software development regarding how to overcome its difficulties and reduce its risk. Amorbieta et al. [2] and Muller et al. [7] discussed how to make a decision on the offshore development, and how to choose right partner, and successfully start, organize, manage and execute this kind of projects.

Moreover, considering the software development, maintenance is a very importance phase, which generally occupies most of the software development time. It is a necessity in order to ensure sound software quality. When an offshore-developed software is used all over the world, it becomes more difficult for an offshore software development team to support the essential maintenance when code development is over.

Nowadays, seldom work studies software maintenance's influence on the offshore software development regarding the issues mentioned above. This raised a number of doubts from our literature study, comparing to our industrial experiences. For example, we believe that the challenges in offshore software maintenance may also affect the project decision-making and execution. How to provide efficient maintenance support in the offshore software development could be a big challenge worth special study.

Regarding the maintenance, some existing results need further study. For example, some work indicated that round-the-clock development is one of the advantages that benefit distributed software development by making use of the time zone difference [4-6]. It is worth further studying whether time difference can really benefit or retard the offshore software maintenance, because we experienced a lot of troubles to overcome the time difference in an offshore software development project that will be studied herein.

Culture liaison was introduced as a great help for alleviating distance and leveraging resources in [3, 7]. Are culture difference and understanding difficulties the only demand for a liaison role? What is the real need in terms of the efficient offshore maintenance? These questions are also worth studying, especially based on real cases.

Generally, the projects that require limited interaction with customers and have low strategic importance and high market capacity are treated suitable for the offshore development in [2, 7]. However, for a global e-commerce project that has more additional challenges than other software projects [1], but developed successfully offshore like the case we will study herein, good approaches used and challenges or lessons learned in its maintenance are especially worth studying in order to extend the theory of offshore software development.

All of above are motivations of this paper. In this paper, we will take a global e-commerce project (GEC hereafter) as an example to analyze the reasons behind its great success and problems/lessons that are worth learning for future offshore software development. The focus will be on software maintenance: how to efficiently support the software maintenance in offshore e-commerce software development.

2. GEC overview

GEC was a web-based service for a global company's partners to order various company products. It was installed at a number of fulfillment sites to support product ordering from any country in the world. It was believed as the biggest B2B e-commerce system in 2000. This system greatly reduced ordering cost, tremendously improved the efficiency of ordering procedure and provided great convenience for both the company and its partners.

GEC provided global automatic management on products ordering, processing and order-maintaining for one of the biggest global companies in the world (customer company hereafter). Its main software was outsource-developed at a company in Singapore (provider company hereafter) during 1998-2001. The provider company completed the GEC software development and maintenance support on totally eight product versions, until the system was very stable and most features' implementation had been done. The system is currently maintained and enhanced by the customer company.

The GEC project was a project executed at different places all over the world. There were totally about fifty persons involved into this project at the provider's company including a development team and a testing team. Figure 1 shows its execution map. At the GEC software maintenance phase, the cooperation among different teams located at different places was needed in order to solve a problem.

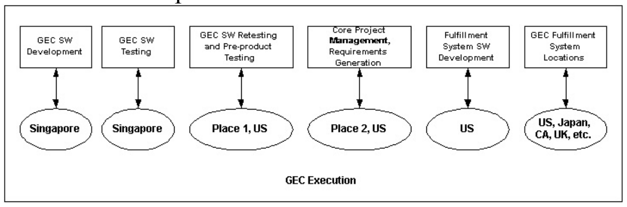

GEC SW Development
GEC SW Testing
GEC SW Retesting and Pre-product Testing
Core Project Management, Requirements Generation
Fulfillment System SW Development
GEC Fulfillment System Locations
Singapore
Singapore
Place 1, US
Place 2, US
US
US, Japan, CA, UK, etc.
GEC Execution

Figure 1: GEC project execution map

The GEC maintenance included several phases after the code development was finished. In this paper, we focus our discussion on the software maintenance conducted at Singapore. Figure 2 illustrates the maintenance phases of the GEC software.

The first phase of maintenance was conducted after the code was built and installed at the local test server in the provider company. The second phase of maintenance was conducted after the local testing passed. The build was uploaded to the testing server located at Place 1, US. The customer company's testing team there retested the software. The third phase of maintenance was started after the build was installed at the pre-product server at Place 1, US. The testing team of the customer company continued the test on more complicated use cases. The forth phase of maintenance was also required if there was any problem found in the product used by the GEC users all over the world. Generally, the product problems were emergent and required to be solved immediately because they directly affected the customer's business.

If there was a problem found in the maintenance phases, the testers either in Singapore or in US reported it using a team-connection tool. The development team could check and reply to the problem report after the fix was done. The team-connection tool managed all problem reports and processing history. Generally, the maintenance work was conducted in parallel with new version's development; especially the third phase and forth phase maintenance.

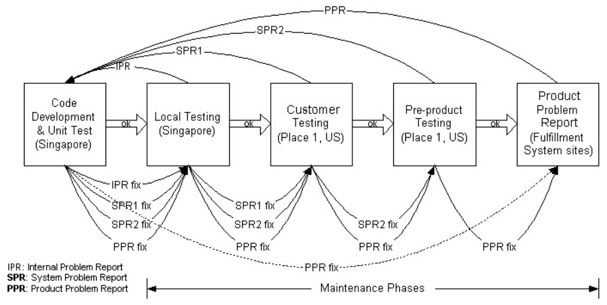

PPR
SPR2
SPR1
IPR
Code Development & Unit Test (Singapore)
Local Testing (Singapore)
Customer Testing (Place 1, US)
Pre-product Testing (Place 1, US)
Product Problem Report (Fulfillment System sites)
ok
ok
ok
ok
IPR fix
SPR1 fix
SPR2 fix
PPR fix
SPR1 fix
SPR2 fix
PPR fix
SPR2 fix
PPR fix
PPR fix
PPR fix
IPR: Internal Problem Report
SPR: System Problem Report
PPR: Product Problem Report
Maintenance Phases

Figure 2: Maintenance phases of GEC software

3. Research questions and methodology

GEC was a successful B2B e-commerce system that brought a lot of benefits for the customer company's global sale. What we intend to study herein is the merits that benefited the GEC maintenance and the lessons that influenced the maintenance efficiency, as well as the aspects worth further improvement. This is because software maintenance support is one of the crucial aspects that influence the whole project's success. In addition, we also aim to clarify the questions mentioned in the introduction through the case study on GEC.

In order to conduct our research, we designed questionnaire and distributed it to all GEC developers for their feedback. The questionnaire was designed based on the first author's personal experience in the GEC development. The first author participated the GEC's development and maintenance on most versions as a component leader. We received pieces of feedback. The questionnaire includes three parts:

- Participator's basic information regarding personal role inside the GEC project and contributions: Based on this part, we can identify the importance of response.
- Factors that affected the maintenance success: We asked the participators to mark the importance of those factors that we thought benefited the GEC maintenance.
- Potentiality for further improvement: In this part, we tried to propose questions in order to study over-time hard work's influence on maintenance efficiency, the reasons of extra maintenance work caused by performance issues, opinions on the difficulties of maintenance support in the GEC, and the reasons that caused the maintenance delay, as well as the hardness of code transference from old responsible person to the new one and from the provider company to the customer company.

Apart from the questionnaire, we also telephony interviewed several GEC developers. These interviewees are software component leaders from whom we can get to know all software components' maintenance information. One of them is the only person experienced all versions' development and maintenance. The main questions asked in the interview are shown in Table 1.

Table 1: Interview questions

1	Which event you experienced in the GEC maintenance gave you deepest impression?
2	What do you think the worst aspects that greatly influenced the efficient maintenance in GEC?
3	What do you think the good methods or approaches used that benefited the GEC maintenance?
4	What do you think the main reasons that delayed or benefited your maintenance work?
5	What is the reason that caused performance issue?
6	What is your opinion on improving the efficiency of GEC maintenance? What are your suggestions?
7	What do you like in GEC? What do you dislike?

In the telephony interview, we tried to get direct feedback on advantages and disadvantages experienced in the GEC maintenance. Especially, we got to know the interviewees' personal opinions on further improvement in order to overcome those bad factors that greatly influenced the maintenance efficiency. Each interview lasted for more than one hour. The interviewees provided valuable answers for each question. Through interview to them, we got a complete perspective on the whole GEC maintenance work.

4. Results

The results we got from the questionnaire and interviews are studied and analyzed as follows.

4.1 Factors benefiting successful maintenance

Based on the questionnaire and interviews, we specified the factors that benefited the GEC maintenance as follows.

It seems that the most important factor for efficient maintenance was attitude and relationship between the development team and its customer. With good attitude and relationship, mutual understanding was easier to build up in order to make trade-off on many issues at both sides. For example, the customer could be easier to understand the reasons of delay on problem solving if they knew the barriers and the hard work at the remote site. The development team would more like to accept extra requirements in urgent and offered solutions as soon as possible.

Compatible development/maintenance environment and efficient communications with the customer were also very important for efficient maintenance. But the provider company's maintenance environment was not perfect to support efficient maintenance. Both testers at US and Singapore did not share the same testing-system Database. This was one of the reasons that made it hard to reproduce the same problem in Singapore, but reported by the tester in US.

The tools such as project pager and approaches (e.g. sending liaison engineer and time-shift work) also played an important role for the efficient maintenance. The project pager was used in project emergency cases. For example, if there was a big problem found in product, the GEC help desk called the project pager. The pager taker at Singapore should call back and get to know the request for urgent maintenance support, even thought at midnight. On the other side, the provider company generally sent a liaison engineer to place 1, US for on-site maintenance support after the first maintenance phase. In addition, a development engineer was also arranged to work at Singapore nighttime, but daytime of US, to support immediate maintenance.

Since the project was big, it was impossible for one person to know all components of the whole system. Generally, it was hard for one liaison engineer and one time-shift developer to solve various problems raised at Singapore nighttime. Generally, they tried to look at the problem, but without any sense to solve the problem. They tried to console the customer until the next morning. They delayed time in order to let the responsible developers have enough time to rest at night, so that they could work efficiently next day. At the same time, they consoled the customer by giving them some feedback to make them feel that the problem was processing at the remote site.

Expert support was also very essential for performance issue. For example, the provider company lacked experts on DB2. The DB2 query caused a lot of performance problems since the database structure is very complicated in order to support the customer's business logic. In order to help the development team, the customer sent a DB2 expert to Singapore. The face-to-face discussion effectively helped the performance tuning work at the maintenance phase.

Other factors that benefited the maintenance work are also important, but those are common factors for both distributed software maintenance and centralized software maintenance.

4.2 Improvement potentiality

Based on the results from the questionnaire and interviews, we summarize the lessons learned from the GEC maintenance in order to seek potentiality for further improvement.

4.2.1 Influence of over-time work. Over-time work was hated by all developers participated in the GEC, especially long-time over-time working (e.g. work from 10am in the morning to 3am next morning for two weeks or work over 3 hours every day for more than one month, which was normal during GEC maintenance). If working over-time, it is impossible to work efficiently and more mistakes may be made because of fatigue. But over-time working was generally forced to do, which happened mostly at maintenance phases if there were urgent problems to fix.

4.2.2 Reasons of extra maintenance work caused by performance issue. Performance issue found later on when the GEC had launched caused a lot of extra maintenance work. This kind of extra work sometimes greatly affected the whole project's schedule. The following reasons were pointed out as importance by the interviewees.

The first reason was that the software designers lacked experience on B2B e-commerce software. They had no much idea which aspects should be paid special attention in the design. GEC is one of the earliest E-commerce applications. It is also among the biggest ones in the world. At that time, no many people held real experience on such kind of software development. In addition, the platform APIs used for GEC development were not mature either.

The second reason was that the software designers at the provider company lacked concrete knowledge on real usage scale and system execution scenarios. Due to tight time schedule required by the customer, performance is not seriously considered at the software design phase.

The third reason was caused by the rapid growth of the GEC usage. The system scale was greatly enlarged within a short period. The initial success also encouraged the customer to deploy this system as broad as possible for its business partners all over the world. This raised many new requirements regarding performance improvement. The dynamic system growth was actually very hard to be anticipated at the design phase.

Herein, experiences were more crucial than technologies in order to avoid performance issue found later in the software product.

4.2.3 Difficulties for maintenance support. Based on the results of questionnaire and interviews, we found that the difficulties of maintenance support were generally caused by long-distance between the customer and the development team. The long-distance made face-to-face communication difficult, which further caused misunderstanding on the business requirements. It also caused time-difference, which, treated as beneficial for round-the-clock efficient software development, actually brought a lot of trouble in the GEC maintenance. The time difference made prompt support on product problem difficult and made it delayed to get feedback from the remote sites.

In addition, the product database was highly confidential. If the product database access was necessary for troubleshooting, the access duration issued was generally quite limited, which made the developers feel big pressure, not mention that the network connection was very slow. Limited accessible machines to the product system sometimes made trouble-shooters have to wait in a queue.

4.2.4 Reasons of maintenance delay. The main reason agreed by most developers about maintenance delay was the difficulties to simulate and re-produce the problem in the local environment. The database data applied for local maintenance were totally different from those for the customer's testing and were obvious distinct from the product. This caused a lot of trouble in reproducing the reported problems. Besides, the execution environment for development was different from the product execution environment. This was another reason made problem simulation difficult.

In addition, the difficulty to exchange idea regarding problems was also an important reason caused the maintenance delay. As commented by an interviewee, he sometimes had to wait until next day in order to get confirmation on some issue. If more discussion needed, longer delay might occur.

Apart from the above, developers' personal reasons and lack of project training might also cause maintenance delay. But those were not treated as so important.

4.2.5 Problems of slow code transference. Due to the frequent resource shifting in IT projects, it is generally impossible for one person to take charge of one software component in all life cycle of a software product. This introduces a practical symptom: responsibility transfer. The slow transfer also affected the GEC maintenance

seriously. This was mainly caused by the following reasons.
1. Lacking formal project training due to tight project schedule: The new responsible person is not so familiar with the project that he/she has to spend longer time to solve the problem.
2. Job competition: The old developer had pressure to be taken over by the new one. So he provided blur technical documents and code comments, and explained the code design carelessly. Similarly, the provider company also faced pressure if the customer withdrew the project.
3. No standard document format, coding format and design pattern deployed: This made new comers difficult to read and understand the code written by other people.

The above reasons also influenced the code transfer from the provider company to the customer's maintenance team after the contract was finished.

5. Discussion

5.1 Managerial implications

Based on the lessons learned from the GEC maintenance, we provide some suggestions for other offshore software projects.

Firstly, it is important design a series of working procedure in order to formalize the project management. It is necessary to make proper project schedule that saves some space for emergent events that may happen later on. Furthermore, it is also wise to make agreement with the customer regarding the solutions on emergency maintenance support, e.g. the accepted rules and policies for additional requirements raised from the product problems. In short, efforts should be made at the contracting phase to evade unnecessary argument that may occur in the maintenance phases. This is also a good approach to avoid hard over-time work that greatly affects the efficiency.

Secondly, it is crucial to pay special attention to performance issue and system scalability in the system software design. It is suggested to invite experienced experts to participate the design on related design issues and make instructions on software development regarding system performance that could guide the developers' programming in general. The customer should provide enough information to its partner about system scalability. It is suggested to provide a paper document to specify the maximum scale of the system, e.g. the size of some database table in the product, the quantity of a normal user's order request. With these approaches, extra performance tuning cost and work could be greatly reduced.

Thirdly, communication problem and time difference raised by the long distance is generally hard to overcome in the offshore software maintenance. It is better to introduce efficient communication tools for easy contact. Many literature studies have proposed a lot of good suggestions on this aspect [8]. But on-line communication or instant message is retarded by the time-difference. If efficiency on maintenance is more important, sending enough technical liaison engineers to the customer site is an effective method. But this may increase the travel cost. Those technical liaison engineers should be qualified enough to handle most of urgent system problems. One liaison engineer is impossible to know every aspect of a big project, so it is impossible for him/her to solve all kinds of problems.

Fourthly, It is better to provide as good as possible equipment to improve the remote access speed for remote problem solving and establish as similar as possible maintenance environment at the local development site. These will benefit problem re-producing.

Finally, It is essential to standardize offshore software project management and organization in terms of efficient code transfer. Project members should be trained for both project general purpose and their personal role purpose. A formalized project document template, coding template and design pattern should be introduced to the project members. This kind of training is a necessity in order to work out uniformed project software.

5.2 Comparison of own results to literature

Maintenance is a very importance phase in the software development. For the offshore software development, the maintenance brings a lot of challenges. Many challenges are actually caused by those advantages that people think could benefit the development according to the GEC experiences, e.g. round-the-clock development actually delays the maintenance; cost saving is normally not true at the maintenance phase because skilled developers are needed to work at the customer site in order to support on-site maintenance. Whilst the development site should also provide maintenance support as usual. The cost is obviously increased if hiring more people. If keeping the same resources, workload will definitely increase that will finally affect the efficiency of maintenance. All of challenges raised by the maintenance should and must be considered when the customer makes decision on outsourcing. The potential extra cost and difficulties that may be caused at the maintenance phase should be seriously considered and calculated at the decision making and contracting stages. Obviously, the maintenance related formal management should be involved into the offshore development management.

Based on our case study, we think it is more challengeable to provide sound maintenance support for globally deployed software product in the offshore development. The issues raised at the maintenance phase are actually ignored in the current literature study. In Table 2, we summarize the research results based on the GEC experiences regarding the maintenance and compare them to the current literature.

Table 2: Research results and comparison to literature

Problems	Good solutions / suggestions	Literature study
Hard to build up mutual understanding to make trade-off on many issues at both sides	The provider keeps good attitude and relationship with the customer (This should be seriously considered at the decision making phase on partner selection.)	N.A. Trustworthy is not considered in [2, 7] for offshore software development
Hard to reproduce the same problem by the development team, but reported by the product users	Set up compatible development/maintenance environment with the product system, prefer as same as possible maintenance environment as the product system; provide sound equipments to access the product system for trouble shooting	N.A.
Maintenance delay caused by time difference	Set up efficient communications with the customer, e.g. making use of project pager for urgent maintenance support; sending enough technical liaison engineers to the customer site for local support; time-shift working in order to provide prompt support	Efficient communication tools are studied in [8]. However, no work proposed technical liaison engineers' great help and time-shift working for software maintenance
Troubles in maintenance raised by performance issue	Invite experts to the provider's site to cooperate with the development team for performance tuning issue; pay special attention to performance on the design; provide as detail as possible scalability description to the provider company; define programming regulations for better performance	N.A.
Hard over-time work that greatly affects efficiency	Design a series of working procedure in order to formalize the project management; consider urgent maintenance issues at contracting and project scheduling	N.A.
Long term code transfer internally and to the customer	Standardize offshore software project management and organization; train project members regarding formalized project document template, coding template and design pattern	N.A.

6. Conclusions and future work

In this paper, the authors studied the maintenance efficiency in offshore software development based on a real case study. According to the questionnaire and interview results, the authors summarized the good points that benefited the GEC maintenance and studied the bad sides that influenced its maintenance efficiency. In order to overcome and avoid those disadvantages experienced in the GEC, the authors further proposed several suggestions that could be referred by similar software development in the future.

Based on the practical experience and the GEC success, the authors believe big global e-commerce project can also be developed offshore although additional challenges need special consideration. The paper proposed some good solutions for potential problems that mostly have not been considered in the literature regarding the maintenance of offshore-developed software.

Since our work is only based on one real case study, the results achieved are only for reference purpose. Future work includes studying a set of efficient maintenance models that can be applied into various distributed software development. It is also significant to define a series of guidelines that could instruct maintenance agreement generation and execution.

References

[1] Alan R. Hevner, Rosann W. Collins, Monica J. Garfield, "Product and Project Challenges in Eletronic Commerce Software Development", *ACM SIGMIS Database*, Volume 33 Issue 4, December, 2002.

[2] Amorbieta, I., Bhaumik, K., Kanakamedala, K. & Parkhe, A., "Programmers Abroad: A Primer on Offshore Software Development", *The McKinsey Quarterly*, No.2, 2001.

[3] Battin, R.D., Crocker, R., Kreidler, J. & Subramanian, K., "Leveraging Resources in Global Software Development", *IEEE Software*, March/April, 2001.

[4] Dedene, G. & De Vreese, J.-P., "Realities of off-shore reengineering", *IEEE Software*, Volume: 12 Issue: 1, Page(s): 35 –45, Jan. 1995.

[5] Herbsleb J.D., Mockus A., Finholt T. A. & Grinter R. E., "An Empritical Study of Global Software Development: Distance and Speed", in *Proceedings of the 23rd International Conference on Software Engineering*, July, 2001.

[6] Mockus, A. & Herbsleb, J., "Challenges of Global Software Development", in *Proceedings of Seventh International Conference of Software Metrics Symposium METRICS 2001, IEEE*, pp182-184.

[7] Muller R., Ruland, D., Hoch, D., & Klosterkemper, B. "Offshore Software Development in Emerging Countries", McKinsey & Company, Articles volumn one – IT Management.

[8] Herbsleb J.D., Mockus A., "An Empritical Study of Speed and Communication in Global Distributed Software Development", *IEEE Transactions on Software Engineering*, Volume: 29 Issue: 6, June 2001, pp481-494.

Risk Management in Global Software Development: A Position Paper

Rafael Prikladnicki, Marcelo Hideki Yamaguti
School of Computer Science, Pontifícia Universidade Católica do Rio Grande do Sul, Brazil
rafael@inf.pucrs.br, yamaguti@inf.pucrs.br

Abstract

The number of organizations distributing their software development processes worldwide keeps increasing, and this change is having a profound impact on the way products are conceived, designed, constructed, tested, and delivered to customers. Global software development exhibits certain features that make it fundamentally different from traditional co-located software development. As the global software development involves additional steps and decisions, these steps also impact the risk management process. The goal of this paper is to discuss some of these impacts and to suggest the development of a process taking into account the dispersion, time zone difference, and cultural boundaries, not only in the operational level, but also in the organizations tactical and strategic level. The paper discussion intends to motivate risk identification, analysis and risk mitigation as earlier as possible in global software projects, foster an efficient risk management process.

1. Introduction

In the last decade, a great investment is being made to convert national markets in global ones. This reality creates new ways for competition and collaboration [1]. However, it also faces some problems like a great number of project faults, and the scarcity of good resources. In this environment, software development organizations found in Distributed Software Development (DSD) an alternative for these problems. DSD is causing a great impact not only in the market, but also in the way the software products are conceived, designed, constructed, tested, and delivered to customers [2]. Sometimes, the search for competitive advantage forces organizations to search for external solutions in other countries, what we call Global Software Development (GSD). In this context, risk management becomes a more sensible activity with a great importance.

Risk, in software area, was represented in a systematic way by Boehm, in the 80's, through the spiral model. This model has as principle to be iterative and risk analysis driven in each iteration [3]. The word "risk" comes from old Italian word "risicare", derived from Latin " *risicu, riscu*" which means "to dare" [4]. In this vision, to run with success to the risk needs more than good processes and intuitive think ability, it needs discipline. This discipline is called risk management.

Nowadays, risk management in software engineering is an evolution of the risk concept that evolved from the analysis in the process model to the management, which should pervade all the processes in the software lifecycle. The risks cannot be just simple details in the project, but they should be the core of the business [5]. Also, risk management have a proactively focus on preventing problems, is continuous, and concurrent.

As the global software development involves additional steps and decisions, we discuss in this paper some approaches to manage risk in global projects, trying to understand the role of all decisions taken in the strategic and tactical levels, and what it represent for the operational level. We call the operational level as the project risk management process, and the strategic and tactical levels all work concerning the decision to develop a project offshore (the long-term offshore road map is part of the strategic level, while the “which center” decision is part of the tactical level).

This paper has the following structure: section 2 presents the impact of GSD on the Risk Management Process; section 3 presents the conclusions, and section 4 presents the reference.

2. Impact of GSD on Risk Management Process

Risk management in GSD is an important and more sensible activity. In a research made by Prikladnicki [2] it was detected that the effective risk management was an alternative to solve existent problems in distributed projects. This is a result from the fact that is hard to deploy, execute and control project in GSD environments because non-technical factors such as social, cultural, behavioral, and political [6], [7]. Other studies [7], [8] also present the same difficulties but due to technical factors such as software development process, project management, project size and complexity.

Therefore, the risk management becomes important in projects that are developed with distributed teams (from the same or different organizations). Besides, independently whether the project is developed globally or in the same city, the fact of having distant teams and using collaboration technologies and developing specific solutions to distributed projects also adds more risk factors to the projects.

In a study conducted by Karolak [9], risk management is part of any project, and risks in GSD projects tend to be more centered in not visible aspects. Also, according to the author, there are three categories of risks in GSD projects: organizational, technical and of communication. Besides, risks belong to more than one category, and these should be in the top of the priorities list.

According to Prikladnicki [2], the risk management in GSD projects should be done not only in the project level, but also in the organizational level. First of all, to decide if a particular project can be developed by globally dispersed teams is difficult (strategic level). Moreover, the decision of where the project will be better developed can also be a problem (tactical level). Some analysis considering the risk and benefit of projects dispersion can be necessary. A number of models are possible and appropriate under different circumstances.

Additionally, it is suggested that all the identified risks in this level should be reflected in the project development level. It means, since the risk analysis was made and the decision of distributing a project was taken, the identified risks must be passed to the project manager. In this way, the project manager can plan response actions to these risks and to add the risks of the whole project, following the risk management process defined for software development.

If we take as an example a multinational organization that has software development units worldwide (offshore software development), some strategies can be implemented. In order to have a better control of the global project allocation and planning, the organization can create a set of activities to be implemented in all projects being developed in the organization unit centers.

These activities involve since the offshore demand definition (strategic level) until the resource allocation (tactical level), what we can call as an Offshore Distribution Model. Once the project is planned and is able to be sent to the offshore centers, the project execution is started, following the organization software development process.

The Offshore Distribution Model can concentrate all strategic and tactical decisions. In this process, a risk and benefit analysis can take place in order to decide if the project can be allocated to an offshore center. Once the decision is made, a risk assessment can be performed having as purpose to verify which center (among all organization offshore centers), can better develop each project. Once the center is defined, all resources are allocated and the project execution can take place, which is part of the operational level, following the organization – or the unit – software development process.

The project execution involves all work concerning the project development by the project team. And this process includes risk management activities. The risk management process in the operational level need to consider all risks identified in the higher levels.

For example, the risk identification activity will search for common risks and past risks in the risk repository, and may involve all project team, including the ones globally dispersed, clients and/or users. Risk identification can consider as one input a document containing information about the risk analysis and risk assessment performed in the strategic and tactical levels (Offshore Distribution Model).

3. Conclusions

Software projects are dynamics and unique, which lead to the existence of many risks that it supposed to be managed. In order to have success in all projects, organizations need to manage risks effectively. But one of the main reasons that risk management is inefficient or it is not implement in many organizations is the lack of documentation of both success and failures in projects. Only the knowledge about risk management is not sufficient.

This paper discussed the role of risk management in global software development projects, considering the strategic, tactical, and operational level of an organization that has implemented GSD. From the point of view of the strategic and tactical levels, organizations can create what we call an Offshore Distribution Model, where a risk analysis and a risk assessment can be performed in order to help in the offshore decision. This can lead to the selection of the best center for a specific project.

Additionally, from the operational level point of view, the risk management in the software development process involves the risk management concerning the project itself. But a key point in the whole process is the integration of the risk analysis and assessment done in the strategic and tactical levels (generally performed by senior managers and offshore centers directors) with the risk management process done in the operational level (performed by project managers). Despite the process to select appropriate centers to develop each project, and a process to manage risks when a project is running, the processes must be integrated to achieve an efficient risk management process for GSD projects.

In short, there are a set of inherent problems and challenges to software development. The GSD, by adding factors like geographic dispersion, temporal dispersion and cultural differences, has accentuated some challenges and added new ones to the development process. Among these challenges we can add as important ones: strategic issues, cultural issues, knowledge management and risks management.

The practice of learning from past experiences, for example, can help senior managers and project managers to plan and control risks [10]. We see a good opportunity to exploit knowledge management benefits, since we are talking about risk management in GSD, which involves some additional steps in the traditional models. And sometimes, risk management in this kind of projects can take longer than in traditional projects, because of the geographic dispersion and time zone difference.

As a result, the work in GSD environments is more problematic than in centralized ones, and the effective risk management can never be depreciated. The risk management importance must be emphasized and its participation must be more decisive in the GSD projects.

Planned follow up studies in this topic will try to analyze some software development units from multinational organizations in order to evaluate the effectiveness of its risk management process, considering all decision levels, and the strategies adopted for global projects.

3. References

[1]. Herbsleb, J. D., and Moitra, D. "Global Software Development", IEEE Software, March/April, USA, 2001, p. 16-20.

[2]. Prikladnicki, R. "MuNDDoS: A Reference Model for Distributed Software Development (in Portuguese)". 145 f. 2003. Master Thesis, PPGCC – PUCRS, Porto Alegre, Brazil, 2003.

[3]. Boehm, B. "Software risk management: principles and practices", Piscataway: IEEE Software, v. 8, p. 32-41, jan. 1991..

[4]. Bernstein, P. "Challenge to God: the risk history (in Portuguese)". Rio de Janeiro: Campus, 1997.

[5]. Kerzner, H. "Project Management: a systems approach to Planning, Scheduling, and Controlling". John Wiley & Sons Inc., USA, 2000.

[6]. Kiel, L. "Experiences in Distributed Development: A Case Study", Proceedings of International Workshop on Global Software Development at ICSE, Oregon, USA, 2003, 4p.

[7]. Carmel, E. "Global Software Teams – Collaborating Across Borders and Time-Zones". Prentice Hall, USA, 1999, 269p.

[8]. Herbsleb, J. D; Grinter, R. "Splitting the organization and integrating the code: Conway's Law revisited". In: ICSE, 1999, Carolina do Norte. Proceedings… EUA, 1999. 11 p.

[9]. Karolak, D. W. "Global Software Development – Managing Virtual Teams and Environments". Los Alamitos, IEEE Computer Society, USA, 1998, 159p.

[10]. Kwak, Y. H.; Stoddard, J. "Project Risk Management: lessons learned from software development", Technovation, In Press, Corrected, 2003

Can Global Software Teams Learn From Military Teamwork Models?

Elizabeth J. Hargreaves, Daniela E. Damian
University of Victoria, BC, Canada
{elizabeth.hargreaves, danielad}@cs.uvic.ca

Abstract

Examining a domain outside of traditional software development may provide opportunities to address the challenges faced by global software teams. In this position paper, we examine the military model since its spirit of cooperative teamwork is well known and clearly documented. Specifically, we explore how an underlying code of conduct and the reinforcing subculture can create highly cohesive, effective teams. Referring to military models in order to build civilian teams is not without historical precedent; we hope that this investigation will prove fruitful. Ultimately, we seek to discover the qualities of the exceptional global software developer while exploring what we believe to be a rich research opportunity.

1. Introduction

A critical success factor for military teams is the underlying ethos that governs the interactions between team members. Dangerous working conditions and high stress levels require these teams to meet positive stereotypes of being honest, hard-working, disciplined and loyal (Feaver and Kohn, 2001). Furthermore, military organizations intentionally develop a distinct subculture to facilitate communication and minimize conflict between individuals from disparate backgrounds, including cultural differences within national boundaries. In contrast, global software development (GSD) teams experience challenges specifically related to teamwork and cultural differences.

This paper intends to stimulate further research into ways in which GSD research can learn from models of military teamwork and which may possibly benefit civilian GSD teams. Note that an examination of the weaknesses and problems inherent to military organizations is considered outside the current scope; in addition, the idealistic nature of this paper is readily acknowledged. For our purposes, we assume that the demands placed on GSD teams differ significantly from those experienced by collocated teams. A GSD team itself is understood to include individuals who rely on computer-mediated communication tools in order to collaborate across significant geographic boundaries.

Military values are typically impressed upon recruits during initial induction and can become an intrinsic part of professional and personal identity. The justification for this philosophy of cooperative teamwork is group survival—with the distinction between the individual versus the group often being ignored. With varying degrees of effectiveness, military organizations coordinate the activities of thousands of people on a global scale and dynamically form new teams on a regular basis. Relying on an established interaction framework that every individual knows, teams can quickly be built from a selection of complete strangers. While this framework may seem impersonal, it can also create a highly productive work environment that prioritizes cooperation over interpersonal politics.

Research shows that GSD teams experience challenges relating to trust, communication, conflict and cultural differences (Damian and Zowghi, 2003; Herbsleb and Moitra, 2001; Oppenheimer, 2002). For example, GSD teams have few opportunities to benefit from the advantages of informal communication. Geographic distances make it harder to establish and maintain interpersonal relationships critical to teambuilding. Subsequently, cross-site negotiations are often characterized by extreme caution in making commitments; in particular, it is harder to trust a remote colleague's arguments, to "see the value of a person" and to anticipate and resolve conflicts at a distance (Damian and Zowghi, 2003). In addition, global teams rarely agree upon communication practices or development processes in which project roles are clearly and well defined at the beginning of the project (Paasivaara, 2003). Many GSD teams operate within corporate environments which thrive on a 'survival of the fittest' mentality with competition between team members often being intentionally fostered by management. Due to this attitude, the 'enemy' can in fact be one's closest team members; as a result, GSD team members may face the additional overhead of protecting themselves from their own team.

In many ways, military teams face challenges similar to their GSD counterparts. For example, naval teams communicate using radio or satellite technologies across huge distances for months at a time with colleagues they may have never met in person. Interactions between coworkers are regulated by a known code since a previously established relationship of trust and accountability cannot be assumed. Undoubtedly, not all military teams function effectively, and sometimes with disastrous results. However, in the interests of productivity and efficiency within the context of GSD, it may be worthwhile to examine the characteristics of successful teams outside the corporate sphere. This exploration is simplistic and not intended to cover the topic extensively; instead, we hope to foster discussion and encourage further research. In the following sections we intentionally consider only two of the potential success factors of military teams.

2. Code of Conduct

Military organizations, such as the American army, rely on codes of conduct as the foundation of teambuilding. Simplistic versions of these codes are typically found in many forms of military literature. The US Army describes itself as: “It's having individual strength and the support of an unstoppable team.” [1] The US Soldier’s Creed places an emphasis on single-mindedness and accountability for teammates as a critical part of the military ethos: “I will always place the mission first./ I will never accept defeat./ I will never quit./ I will never leave a fallen comrade.” [2] How closely the code is followed is, in a sense, a measure of the level of professionalism achieved. Despite an uncertain level of confidence in the military overall, Americans continue to consider their soldiers to be the most highly respected professionals in the country (Feaver and Kohn, 2001).

While professionalism is undeniably important within corporate spheres, the corresponding conduct is often ambiguous and can change dramatically based on context. Shifting mores within the field of GSD can be particularly problematic when faced with the previously mentioned challenges of reduced trust and ambiguous communication. Furthermore, the professional responsibilities of software developers remain in embryonic form since a comprehensive code of conduct for software developers is still developing. Personal reputations are frequently based on technical expertise as opposed to an ability to ensure the success of fellow team members or a high level of personal integrity. High turnover and unstable markets no doubt also contribute to shifting allegiances and a diminished sense of loyalty.

3. Military Subculture

Military codes of conduct are reinforced by the surrounding subculture. Stripped of the financial and professional incentives found in corporate environments, soldiers have fewer motivations to work against one another. Known pay scales and the rigidity of the rank system do not provide an equivalent opportunity for advancement and reduces competition among peers. Furthermore, the vertical chain of command and a visible hierarchy simplifies communication between coworkers. Informal communication is also highly influenced by this subculture—slang, jokes and topics commonly discussed within the military environment contribute to creating cohesive teams.

In contrast, corporate environments provide a lot of opportunity for ‘leapfrogging’ over colleagues while corporate secrecy permits negotiable salaries. Invisible hierarchies within corporate environments (exacerbated in global software teams) often result in personnel devoting a significant amount of time negotiating political minefields instead of working productively. While a software subculture certainly exists, interactions with others are not necessarily based on principles of trust and integrity, nor is there a consistent level of personal accountability for other team members.

4. Conclusion

In this paper we encouraged approaching current challenges in global software teams by learning from the critical success factors found in military teamwork models. We believe that cooperation is an undeniably critical dimension of GSD and suggest that a reinforced sense of teamwork may enable team members to overcome GSD challenges. Specifically, we seek to examine how a defined code of conduct and supporting subculture may allow team members to overcome problems related to trust, communication, conflict and cultural differences.

The challenges faced in GSD are not unique from an organizational perspective. Referring to military models to build civilian teams has significant historical precedent. Police, paramedical and fire-fighting units are examples of civilian organizations that successfully leverage military techniques in order to build successful teams. Can GSD teams use these same techniques? Note that we are not trying to create a platoon of programmers; instead, we wish to simply adopt the positive traits found in military teams. In addition, we do not expect that the same level of discipline found in military environments would be necessary within the software domain. Finally, if military teambuilding techniques are successfully adopted, is there a research opportunity to develop tools and methodologies to support it? Ultimately, we seek to identify the characteristics of the exceptional global software developer. We also strive to determine how to develop and nurture these same traits in individual developers in order to build highly effective GSD teams.

5. References

Damian, D. and Zowghi, D. "RE challenges in multi-site software development organizations", *Requirements Engineering Journal,* 2004.

Feaver, P.D. and Kohn, R.H. "Uncertain Confidence: Civilian and Military Attitudes About Civil-Military Relations", *Soldiers and Civilians: The Civil-Military Gap and American National Security.* MIT Press, Cambridge, 2001.

Herbsleb, J. and Moitra, D. "Global software development"*, IEEE Software,* March/April, 16-20, 2001.

Oppenheimer, H. "Project management issues in globally distributed software development", Proc. International Workshop on Global Software Development, 2002.

Paasivaara, M. "Communication Needs, Practices and Supporting Structures in Global Inter-Organizational Software Development Projects", Proc. International Workshop on Global Software Development, 2003.

[1] "What is the US Army?" GoArmy.com – Army 101. United States Department of Defense. http://www.goarmy.com/army101/index.htm. Accessed Mar 21 2004.

[2] "The Soldier's Creed". Our Army At War – Relevant and Ready. United States Department of Defense. http://www.army.mil/thewayahead/creed.html. Accessed Mar 21 2004.

Designing the Inter-Organizational Software Engineering Cooperation: An Experience Report

Hans W. Nissen
RWTH Aachen, Informatik 5, Ahornstr. 55, 52056 Aachen, Germany
nissen@informatik.rwth-aachen.de

Abstract

This paper reports about experiences in managing the transformation from internal development and maintenance of software engineering tools towards an external one. We describe three different inter-organizational cooperation forms which differ in the distribution of development responsibilities between client and vendor – and which support the distributed design of three different classes of software products. An important finding was that even for software engineering tools which were extremely important for project success a carefully designed relationship model enables a successful distributed development.

1. Introduction and Background

This paper reports about experiences in establishing an inter-organizational cooperation between a world-leading telecommunication company and Indian IT service providers. The experiences we report focus on the selection of cooperation forms and the necessary organizational changes, but we are not looking at any financial aspect.

The subject matter of the cooperation was the huge set of proprietary software engineering tools used and developed within the telecom company. The cooperation goals from the customer perspective were to save money, to keep product quality and delivery precision, and to start a long-term cooperation with an external service provider.

Tools. The specific character of the software engineering tools influenced the selection of the form and the organizational set-up of the cooperation to a large extend. Therefore, we give in the following some background information on this subject matter.

Many products of the telecom company were based on a proprietary hardware platform and programming language. As a consequence, all supporting software engineering tools regarding that platform have been developed in-house during the last 20 years – all together the tool suite contained about 200 tools. Based on the companies strategic plans for software and system engineering, these tools were classified into three categories:

- *project critical tools which require continuous development and improvement activities*; examples are: compiler, simulated test environment, build environment. An erroneous or delayed tool delivery would cause major problems for the target projects developing telecommunication software.
- *tools which require further development activities*, but the new functionality is not project critical (i.e. the project could also survive without the development); examples are: version control system, modelling environment, traceability tools
- *tools which do not need further development activities but only maintenance activities* (i.e. bug fixing to some extend); examples are: editors, database applications to coordinate resource usage (e.g. error codes, signal names), document management system, fault tracking tools

Processes and Roles. The in-house communication and development processes were based on a formal client-vendor model: product managers search for (internal) project sponsors, collect requirements and develop the product release strategy. However, the internal tool development departments acted very flexible and accepted late and major changes to the original requirements. In case of faults and improvement suggestions, a fast and direct communication between tool users and tool developers was always possible. It was even seen as an advantage that this short feedback loop enabled high quality products. Especially for the project critical tools the direct and informal communication guaranteed the required quality and delivery precision.

2. Selection of Cooperation Form

The goal with setting-up an inter-organizational cooperation was to outsource a large part of the software engineering tool's development processes to external vendors under the constraint that neither the quality nor the delivery precision must suffer.

The literature proposes cooperation models which differ in the distribution of development activities between client and vendor site. The outsourcing project analysed and discussed the following three cooperation models (cf. figure 1).

Classical contract model: The client delivers a set of specifications and the vendor implements or updates the software accordingly.

This model was selected to handle the non-critical tools without further development activities, i.e. tools which will go into maintenance mode. In this specific set-up the client does not deliver requirements for new functionality but requirements on fault corrections.

Implementation model: The client keeps much responsibility in-house: He specifies the requirements, identifies the impacts on the system components and produces the updated system and component specifications. The vendor is doing the detailed design on unit level and does the basic test. The integration test as well as the acceptance test is again performed by the client. This model leaves much intellectual work and responsibility with the client while the tasks of the vendor are limited more or less to the implementation part. This model requires good programming skills but less system management skills at vendor site.

This model was employed for the project critical tools because the control of these tools should remain within the company - at least the beginning of an inter-organizational cooperation.

Product management model: The vendor is taking over parts of the product management activities while the ownership stays with the client. The client participates in requirements specification and performs the acceptance test; the remaining parts of analysis, design, implementation and integration test activities are all done by the vendor. This model pushes most of the responsibility and work to the vendor and requires a very good understanding of the software product and the system environment at vendor site.

This model seemed to be most suitable for the non-critical tools with further development activities. The client company reduces the costs as much as possible and the vendor gets a chance to handle a product from beginning to end and to prove its competencies.

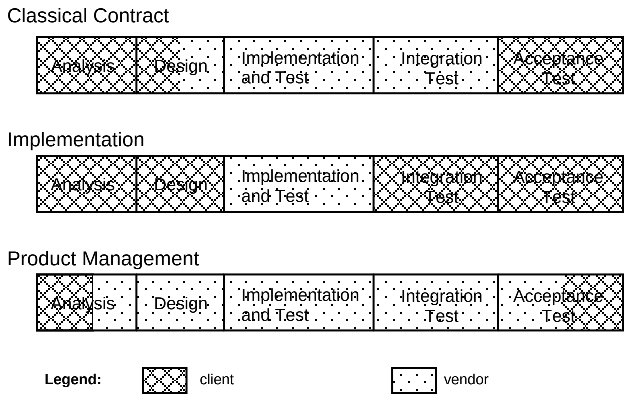


Figure 1. Three cooperation models

These different cooperation models require also different kinds of client-vendor relationships. The FORT framework described in [2,1] was identified to be valuable to characterise the resulting relationships. The FORT framework consists of two dimensions relevant for inter-organizational co-operations. The first dimension deals with the extent of ownership or control substitution by a vendor. The second dimension deals with the strategic impact of the portfolio. The four resulting types of cooperation relationships are support, alignment, reliance and alliance (cf. figure 2).

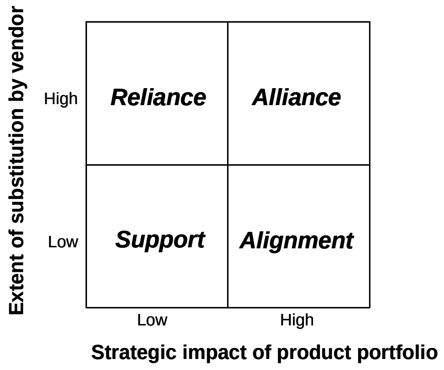

Extent of substitution by vendor
High
Low
Reliance
Alliance
Support
Alignment
Low
High
Strategic impact of product portfolio

Figure 2. The FORT Framework [2,1]

In the support relationship, both the extent of substitution and the strategic impact of the outsourced products was low. The role of the external provider is therefore very limited. Within the alignment relationship, the substitution is low but the strategic impact of the outsourced products is high. This relationship is often used to obtain a service provider's specific expertise for a project. The reliance relationship highly involves the service provider in the client's processes. It therefore requires a high level of commitment from the vendor and a high level of reliance into the vendor competences from the client. Finally, in the alliance relationship the vendor takes over a major part of the responsibilities for a product of high strategic value for the client. The two parties act as strategic partners with common goals [2,1].

The telecom company applied the product management model for the products with a medium strategic impact and included a high extent of substitution. This technical cooperation model is therefore best supported by the reliance kind of relationship. The implementation model on the contrast includes a low extent of substitution but is employed for products with a high strategic or risk impact. Therefore, this cooperation model requires an alignment kind of relationship. Finally, the classical contract model was designed for products with a low strategic impact and includes a medium extent of substitution. It can be located within the support relationship with a tendency towards the reliance relationship.

Summarising the selection of cooperation forms, the

- *project critical tools* were handled in an alignment relationship using the implementation model for distributing the responsibilities
- *non-critical tools with further development activities* were handled in a reliance relationship with the product management model
- *non-critical tools with no further development activities,* i.e. tools which go into maintenance mode, were handled in a support relationship with the classical contract model.

3. Changes in the Organization

The establishment of the cooperation with an external organization regarding the development of the software development tools led to some changes – in processes, roles and attitudes – of which we will mention only a few.

Processes and Attitudes. Within the requirements engineering process the developed specifications became more formal and the process itself was followed in a much stricter way. The habit of including late requirements into a development project – and sometimes changing the directions of the project by this completely – was abandoned. The product management process includes now as well the definition of formal acceptance test cases which did not exist before. These test cases and the requirements form an important part of the contract for product improvement; late changes would therefore cause unpleasant re-negotiations – in contrast to the flexible company-internal handling of such changes.

As new instances, monitoring processes have been installed. For the classical contract model, a simple variant in form of measurements on the delivered products has been chosen. For the two other cooperation models, a more sophisticated monitoring concept has been developed. Progress and product quality is measured during the whole development process, at clients and at vendor site.

Roles. The roles of product manager and system manager for the SW development products changed from operating only internal to acting as interface towards the vendor. From the vendors perspective the system manager become the most important role within the implementation model. He followed the implementation and answered the questions. At least in the first projects the goal was to exchange personnel in the way that ain the early phases employees of the vendor works together with the system manager at the client site. In later project phases the system manager will work partly at the vendor site to support and monitor the implementation activities.

The product manager is a key person for the product management model since he supervises the requirements acquisition and specification activities and takes the full responsibility from the client side.

Technical Environments. All the software development tools are integrated into a tool suite. This suite is still produced in-house whereby the integration test is the most important activity. This test verifies the programming interfaces, the external procedure calls, the side effects and the required documentation. The cooperation with external vendors required the external availability of the integration test suite such that the vendor himself is able to verify his tools. Within all cooperation models, faults reported by the users have o be forwarded to the maintenance department. The reporting and tracking environment used so far only in-house had to be available for external vendors, with the requirement that internal knowledge and information is really kept internal.

4. Conclusions and Lessons Learned

The establishment and the operation of the inter-organizational cooperation was successful regarding the stated goals, namely to save money by outsourcing the tool development and maintenance under the condition that neither the quality nor the delivery precision must suffer.

For the three different groups of tools three different cooperation models and relationship types have been implemented. The changes to some parts of the organization have been high regarding processes, roles and attitudes.

The lessons learned in this project can be summarised as follows:

- The applicable relationship types to the external vendor and their impacts to the organization must be carefully studied before a contract is signed or existing internal departments are discarded.

- There are major changes to the product management role and the requirements engineering processes and attitudes when moving from internal software development towards a global development model.
- An effective and honest change management is essential to successfully establish an inter-organizational cooperation which effects on employees.
- The establishment of an inter-organizational cooperation should be organised as a project with a set of clearly defined decision points, at least for the set of considered products, the designated cooperation forms, the resulting changes and their implementation strategy, the selected service provider. The management and the technical experts must be involved in all of these decision points.

References

[1] R. Kishore, H.R. Rao, K. Nam, S. Rajagopalan, and A. Chaudhury, “A Relationship Perspective on IT Outsourcing”, CACM, December 2003, Vol. 46, No. 12, pp. 87-92.
[2] K. Nam, , S. Rajagopalan, H.R. Rao, and A. Chaudhury, “A two-level Investigation of Information Systems Outsourcing”, CACM, July 1996, Vol. 39, No. 7, pp.36-44.

Acknowledgments

This experience report was funded in part by the German Ministry of Education and Research, BMBF, in project “VSEK” (FKZ 01 IS C65).

Towards a Model of Awareness Support of Software Development in GSD

James Chisan, Daniela Damian
Department of Computer Science
University of Victoria
PO Box 355, Victoria BC V8W 3P6 Canada
{ chisan, danielad } @cs.uvic.ca

Abstract

Awareness is a powerful concept that can be used to enable developers to quickly and easily grasp the state of the workspace which they operate within. We begin by explaining one approach to how awareness might be used to support software development and to enhance developer cooperation and communication. However, since this approach assumes on-going collaboration it is useful to couch the discussion within a collaborative model. This paper presents a model of how awareness could support software collaboration, by describing typical software collaboration, how it is deficient and how awareness helps ameliorates those deficiencies.

Finally, we discuss a variety of issues that become apparent when considering the approach. It is intended that these issues will stir debate and may help generate insight that ultimately improves global software development via awareness support.

1. Introduction

Software development is essentially a collaborative effort among the stakeholders of a project, especially so for those directly involved in development. Business analysts, system analysts, designers, programmers and testers all work together toward the common goal of producing a software solution. Furthermore, they do this in a common workspace comprised by the intermediate artifacts of development: requirements document, design, test scenarios, code, etc. This paper describes a model for how awareness can support collaboration during software development. In global software development (GSD) environments physical separation impairs communication and infringes on the collaborative freedom collocated developments enjoy. Therefore the aim here is to illustrate our vision of how awareness of the workspace might address deficiencies in software collaboration that are exacerbated during GSD.

This paper follows up on work presented last year at the ICSE 2003 GSD workshop (Damian, Chisan, Allen, and Corrie, 2003), where we described how [small] co-located teams benefit from social mechanisms that naturally facilitate work practices and diminish the apparent need for explicit workspace awareness support. To address this need, we propose that when changes are made within the workspace environment particularly to requirements to which much subsequent development depends, developers should be selectively notified about the nature of the change so that changes in requirements are quickly integrated into their work and the development effort on the whole. Requirements are particularly important since it is here that important decisions which directs all subsequent development is, or should be, recorded.

This work describes the first step to implementing that proposal. It seeks to illustrate a model that shows how awareness can support existing collaboration patterns that are typically exhibited during software development. By using the model, shortcomings in collaboration practices that occur during development can be identified. Then, the means in which awareness can support such practices is demonstrated and analyzed. Ultimately this serves to contribute to the larger, primary research goal to improve software development collaboration.

2. Background

Awareness simply refers to knowledge one has of the changing environment which one operates within, essentially ‘knowing what is going on’ (Endsley, 1995). In software development, this environment is the workspace environment composed of all the intermediate products of development. In particular, with respect to requirements engineering, document change and contact discovery, both facets of the workspace, have been identified as a source of ineffectiveness, confusion and development paralysis (Herbsleb, Mockus, Finholt, and Grinter, 2000). Furthermore, development is hampered by ‘organizational amnesia' where issues that have already been discussed and resolved are repeatedly reopened for no other reason than their outcomes have been forgotten (Catledge and Potts, 1996). While the current state of the workspace could be ascertained from the contents of the organizational and project documents, these documents have been shown to be a poor communication media (Curtis, Krasner, and Iscoe, 1988; Al-Rawas and Easterbrook, 1995). Likely, as in the case of requirements, formal mechanisms (documents) are not updated quickly enough and, instead, news is propagated informally (Herbsleb *et al.*, 2000). As developers come to depend on informality, it is no wonder there is little motivation to record progress in formal documents in a timely manner.

While there are a variety of tools that implement awareness as an central feature of the system (Jang, Steinfield, and Pfaff, 2000; Bentley, Horstmann, Sikkel, and Trevor, 1995), these systems are not tailored directly to software development. In some cases, even when such tools are used primarily in development support, awareness refers only to notification of change to authors of the artifact (Brush, Bargeron, Grudin, and Gupta, 2002). Such an approach is not sufficient where developers rely on documents they do not author themselves.

In contrast, our proposal is to use the artifacts that exist in the workspace and from their contents build, [semi-] automatically, relationships describing document hierarchies and their authors. This establishes dependence between artifacts (ie. design *x* fulfills requirement *y*) and developer-artifact relationships (ie. *Jim* and *Bob* wrote design *x*). Then, when changes in artifacts in the workspace are detected, authors of dependant documents are notified (ie. *Jim* and *Bob* are notified when requirement *y* changes). This method serves to leverage existing document structures and content to selectively deliver awareness to those who need it.

3. Purpose of the Model

Software development is largely a collaborative task, a result of many different stakeholders working closely together to implement a software solution. The purpose of the model presented in this paper, is to show how awareness fits in to the broader system of development collaboration. This is necessary to articulate, in a structured, detailed way how development practices might be improved with awareness support.

To improve development practices, any method must consider the constraints which limit the possible solutions. In large part, the development habits and processes within organizations are a severe constraint that any approach must consider. Thus the model is used first to describe collaboration patterns that capture the nature of current development practices as a means to show potential insufficiencies in these practices. Second, the model serves as a means to structure where faults occur in collaboration and to analyze the nature of awareness support that might address these faults and strengthen the effectiveness of collaboration. By utilizing this model, we establish a formal descriptive, theoretic framework on which to base further research.

4. The Model

To show how failures in software collaboration might be improved with awareness support, we begin by describing an idealistic model of collaboration and how it is weakened in GSD projects. These weaknesses transcend purely GSD origins, such as impaired communication, lack of informal, impromptu discussions; they also relate to other pressures such as time to market, resource constraints or skill shortage.

To accurately describe typical projects that might involve GSD, a development effort of sufficiently large magnitude must be chosen. Therefore, it is assumed that there are a set of unique stakeholders in the project that include: business interests of (1) the development company and (2) the customer company, (3) end users, (4) system analysts, developers including (5) designers, (6) programmers, (7) testers and (8) documenters. For completeness it is useful to briefly illustrate the roles of these stakeholders.

The business interests of the development company may include marketing tactics, product strategy, future vision, stockholder interests, internal efficiency and policy. The customer company is responsible for the contractual and financial obligations of the software purchase, thus their concerns may be primarily more basic focusing on cost, benefit, product adoption and product support. However, the goal of software is to improve the efficiency and effectiveness of the end user whose satisfaction is based on usability, features, quality and reliability. System analysts are tasked with the responsibility of determining the characteristics (requirements) of the software solution, typically from the above mentioned stakeholders, while being constrained by the limitations of the following development stakeholders. Development consists of designers who elaborate on system requirements and produce detailed technical designs to satisfy software solution, programmers use those designs to produce software that complies with their design, meanwhile testers use requirements and designs to develop, and later execute, test cases. Documenters also use requirements and design to publish user documentation.

As figure 1 illustrates the model shows collaboration paths as work on the software development occurs in clusters of activity each centered on task types within the development. While many interactions occur throughout iterations within the activity (circular), equally important inter-

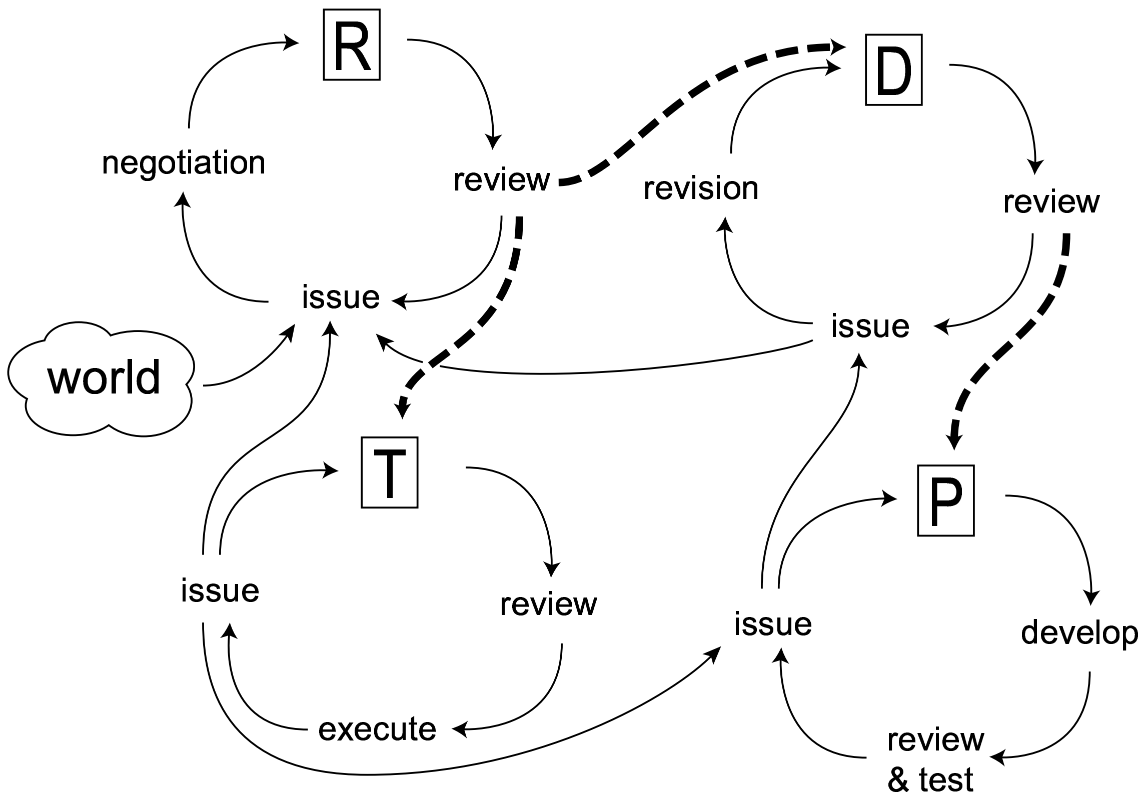


Figure 1. Model of collaboration in software development. Bold, dashed arrows show collaboration paths between development tasks that can be supported with our awareness approach

actions occur between activities. System analysts develop requirements ("R") then review requirements and field concerns about requirements from development and incorporate issues raised by non-development stakeholders, such as customers, users, management or marketing. The requirements activity is concentrated around the task of developing concise, complete requirement documents. Designers take requirements and develop technical designs ("D") then they refine their designs by raising concerns about ambiguous, conflicting or impossible requirements, and by fielding concerns of programmers. The design activity is concentrated around the task of developing complete, logical designs for programmers. Tests also take requirements (and may also use designs) to develop test cases and test scenarios to validate the software ("T") then they refine their tests by raising concerns about questionable requirements and by coordinating with programmers over the execution of their testing procedures. The testing activity is concentrated around the task of developing tests to validate that the software product works and fulfills the software requirements. Programmers take designs and develop the code that becomes the software product ("P") then they progressively refine their code by raising concerns about the design to designers. The programming activity is concentrated around the task of developing functional software that satisfies their designs. In ideal circumstances once the requirements activity has concluded no further events would affect the nature of the requirements, design and test could begin, and then at the conclusion of detailed design, programming could begin and finally the software would be validated with a minimum of activity iteration.

Unfortunately, the model fails to capture the difficulties encountered during real-life software development. In all development the constraint of time bears down most heavily on the life-cycle of the development process. Development that occurs in GSD environments is further handicapped by a lack of timely, lucid communication between developer activities.

When time is constrained, development time is compressed by encouraging each activity phase to begin as soon as possible. In many cases this means that requirements, design, coding and test all start simultaneously. However, the consequence of this philosophy is that interactivity communication is increases, both in frequency and importance.

As requirements become available it is crucial that they are communicated quickly to development and test to minimize the efforts they spend using out-of-date requirements information. (see Figure 1, bold arrows) Likewise, designs need to be propagated quickly to programmers so that they minimize wasted effort. Conversely, issues about requirements need to be recognized quickly by design so that systems analysts can resolve conflicts and ambiguities. During GSD, this scenario is further hindered by slow, sometimes asynchronous communication that is unclear and ambiguous.

Software development failures have often been attributed to requirements error. In particular, captured changes in requirements are not broadcasted to appropriate developers who would otherwise adjust their efforts to reflect such change. In part, this may be attributed to requirements that are only informally captured and are not formally articulated within requirements documents, which become, and remain, habitually out-of-date - making them a redundant waste of effort (Herbsleb *et al.*, 2000).

For example, consider the following frustrating scenario that many industrial practitioners could probably identify with: An unavoidable technical constraint discovered by a programmer causes design to be reworked, designers negotiate an adjustment in requirements with system analysts and modify their designs appropriately. Subsequently, this technical constraint is raised repeatedly by other programmers who were not informed of the design changes (or are using designs that may have been unrelated to the adjusted design, but related to the affected requirement). This chaotic disruption wastes time and effort and causes unnecessary aggravation to the development team.

Clearly this scenario could benefit directly from an awareness of changes in the development workspace that reflect project decisions made by developers. By effectively employing awareness for development artifacts, it is possible to automate some of the communication that occurs between development activities. (see Figure 1, bold arrows). By detecting changes within requirements and automatically notifying relevant designers, testers and programmers, these developer stakeholders can be kept aware of the requirements on which they rely. Furthermore, this promotes development artifacts as first class documents in which to record and organize information. Developers are not interested in wasting their time polling documents for change, but if notified of changes, they are (further) motivated to refer to those documents to determine the nature of those changes. Authors, system analysts in the case of requirements, are motivated to articulate their refinements in the document. Thus, the document itself becomes a medium of communication and developers can begin to rely on their timeliness and currency.

Although this approach does not address the effectiveness of any particular activity, it does suggest that improvements to interdepartmental communication can be realized. For example, awareness does not help the system analyst to capture shifts in markets or abrupt changes business strategy. In contrast, awareness helps the system analyst to communicate these changes (once identified) via the requirements document in a reliable timely fashion to relevant developers. Only then can development be made dynamic and responsive to emergent requirements inherent during iterative, time-constrained development.

5. Issues of the Proposed Solution

In using the model to develop possible awareness solutions to improve collaboration and development during software development a variety of issues become apparent. The issues described below are presented for discussion during the workshop in which this paper is submitted, in the hope that further insight can be exchanged by attendees.

5.1. Extent of Information Dissemination

Of central concern to providing awareness to participants of the development process is striking a balance between providing information germane to their current work responsibilities and limiting extraneous, redundant information that overwhelms developers or desensitizes them to awareness mechanisms. Rather than a scattershot broadcast approach, the analysis of person-artefact relationships makes it possible to provide information to developers in context to their responsibilities and contributions to the workspace.

To maximize the accuracy of notification, person-artifact relationships can be, in many cases, extracted from existing information in the workspace - for example authors of a design could be extracted from the design itself. While this may establish sufficient relationship between developer and artifact additional questions still remain about the extent of awareness required to keep developers up to date.

5.2. Privacy

Software organizations interact in increasingly complex arrangements of acquired, partner and/or outsourcing companies. Data that is [automatically] collected from intermediate artifacts created during GSD can span physical and organizational boundaries. If this information is used for awareness purposes, then this represents a potential for information flow across boundaries, in such cases questions of privacy may arise. For example, an outsource company may want to limit what information is be collected and disseminated to its client (an intimate stakeholder in the project). Even within a single company, some developers may oppose the collection of information that could reflect on their progress and productivity. This issue can be most closely related to that of personal privacy problems that have been considered with respect to video conferencing (Boyle and Greenberg, 2000) and presence awareness (Godefroid, Herbsleb, Jagadeesany, and Li, 2000), where control over data fidelity is used to maintain presence awareness while preserving personal privacy. This issue differs because it transcends personal privacy to include organizational privacy, and is with respect to the virtual workspace rather than a mere reflection of physical space.

5.3. Delivery

Ideally awareness of the workspace should be as natural as awareness of night and day. The challenge is to minimize the effort and distraction required of developers to keep up-to-date with the goings-on within the workspace. In other words, to provide awareness information as tacitly as possible. Delivery of awareness can vary widely although a few obvious choices include the provision of awareness information in a textual or semi-textual/visual manner via email, the web, instant messenger, or with the use of some specialized application. Management overhead, more stuff to read

5.4. Visualization of Artifact Relationships

Providing awareness requires the establishment of relationships among artifacts and between artifacts and the stakeholders involved in the development. Once this information has been collected it may have significant value on its own as a resource for developers and analysts. Furthermore, these relationships may dramatically enrich awareness events delivered to developers, providing context for the event. The inherent value of this information suggests that it needs to be intuitively accessible to developers. Naturally, we wish to consider how these relationships could be visualized by determining what information developers need to extract and what sorts of questions they may find themselves asking about relationships among artifacts.

6. Future Work

The model presented above is only an intermediate step on the way to achieving the larger research goal of improving software development by providing better support to collaboration in global software teams. This model is a first version based primarily from reports reviewed in available literature, however validation of this model is required. To validate the model we intend to present the model to an industrial partner and survey a sample breadth of stakeholders to critique the model based on their experience. With this input the model will be adjusted to capture those experiences and through this validation process the development of the model will continue to evolve, becoming more refined and more accurately capturing the true nature of collaboration during software development. In the longer run, the model will be used to develop technological or process-based solutions that are specifically designed to deliver awareness to enhance collaboration in GSD. Shortcomings of that solution will also serve to reflect insufficiencies of the model so that the model can be improved.

7. Conclusion

Awareness is a collaborative response to the problem of improving software development practices in GSD. The model described herein serves to show how awareness could be used to improve the naturally collaborative process of software development. Not only could awareness help ameliorate GSD-specific issues, but such solutions also promise to be highly beneficial to co-located development too.

References

A. Al-Rawas and S. Easterbrook. "Communication problems in requirements engineering: A field study." In *First Westminster Conference on Professional Awareness in Software Engineering*. London, 1995.

R. Bentley, T. Horstmann, K. Sikkel, and J. Trevor. "Supporting collaborative information sharing with the WWW: The BSCW shared workspace system." In *Proceedings of the Fourth International WWW Conference*. Boston, MA, 1995.

M. Boyle and S. Greenberg. "Balancing awareness and privacy in a video media space using distortion filtation." In *Proceedings of the Western Computer Graphics Sumposium*. 2000.

A. J. B. Brush, D. Bargeron, J. Grudin, and A. Gupta. "Notification for shared annotation of digital documents." In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press, 2002.

L. Catledge and C. Potts. "Collaboration during conceptual design." In *Proceedings of the 2nd International Conference on Requirements Engineering*. 1996.

B. Curtis, H. Krasner, and N. Iscoe. "A field study of the software design process for large systems." *Communications of the ACM*, 31(11) 1988, pp. 1268–1287.

D. Damian, J. Chisan, P. Allen, and B. Corrie. "Awareness meets requirements management: awareness needs in global software development." In *International Workshop on Global Software Development (colocated with ICSE '03)*. Portland, OR, 2003.

M. Endsley. "Toward a theory of situation awareness in dynamic systems." *Human Factors*, 37(1) 1995, pp. 65–84.

P. Godefroid, J. D. Herbsleb, L. J. Jagadeesany, and D. Li. "Ensuring privacy in presence awareness: an automated verification approach." In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM Press, 2000.

J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. "Distance, dependencies, and delay in a global collaboration." In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM Press, 2000.

C. Y. Jang, C. Steinfield, and B. Pfaff. "Supporting awareness among virtual teams in a web-based collaborative system: the teamscope system." *SIGGROUP Bulletin*, 21(3) 2000, pp. 28–34.

Peer-to-Peer Remote Conferencing

Fabio Calefato, Filippo Lanubile, Teresa Mallardo
Dipartimento di Informatica,
University of Bari
{calefato | lanubile | mallardo}@di.uniba.it

Abstract

Global software development (GSD) is nowadays pervasive among large enterprise organizations. Physical separation in GSD has raised many issues, mainly due to cross-sites communication and coordination problems, which have made software development an even more challenging task. Hence, distributed workgroups need tools to support a load of activities that usually take place through the direct interaction among people. This paper presents a tool, called P2PConference, to conduct conferences over a distance. The tool provides basic features for simple brainstorming sessions as well as more sophisticated features to accommodate the needs of other types of meetings, such as presentations and panels. P2PConference adopts a decentralized architecture and it is implemented upon a peer-to-peer infrastructure platform, called JXTA.

1. Introduction

Over the last few years, large enterprise organizations have embraced global software development distributed over multiple geographical sites [9]. Communication is the core function of cooperation that allows information to be exchanged between team members. Distance has a negative effect for communication-intensive tasks, such as software design, and on spontaneous conversation [8], where people informally communicate valuable pieces of information.

Distance is usually offset by Internet-based technologies: globally distributed workgroups typically rely on centralized systems, mostly built on top of web-based development platforms, to support collaboration across time and space. However, peer-to-peer (P2P) applications, based on a decentralized architecture, are increasingly becoming popular to exchange instant messages, share common information and applications, and jointly review/edit documents. Collaborative P2P applications exhibit the following advantages with respect to client-server counterparts:

- Autonomy. In a P2P system every peer is an equal participant while being a final authority over its local resources. In this way everyone can share information but, at the same time, can pose restrictions on confidential data through access rights management and data encryption. When enterprise data are distributed on many places and on different devices, P2P systems can provide an easier and cheaper alternative to enforcing a convergence into a centrally managed data repository.
- Intermittency. P2P systems are designed by giving for grant that any peer can disappear at any time because of network disconnections, either deliberate or accidental. P2P collaborative systems use resource replication and different synchronization mechanisms, based on proxies for sending/receiving messages in the network on behalf of the disconnected sender/receiver. In this way, users can work to shared content even when offline and automatically propagate changes at the first reconnection.
- Immediacy. P2P applications have shown themselves able to support direct exchanges between peers, as in the case of instant messaging. P2P collaboration systems, based on near real-time communication mechanisms and synchronous presence of the peers, can provide immediate responses by participants to enable effective person-to-person interaction.
- Cost lowering and compartment. P2P systems are valuable means to lower infrastructure cost by using existing infrastructure and distributing the maintenance costs. Centralized systems that serve many clients typically bear the majority of the cost of the system. When the cost becomes too large, a P2P architecture can help spread it over all the peers.

Under these conditions, a P2P collaborative infrastructure can complement or even replace client-server platforms for the creation of ad-hoc or small workgroups, drastically reducing the cost of infrastructure setup and ownership. Due to P2P own features, it is possible to quickly establish dynamic collaborative groups, composed of people from different organizations accessing shared resources and interacting in a near real-time manner.

This paper presents P2PConference, a P2P remote conferencing tool which has been developed at the University of Bari. In the next sections, we first introduce the underlying platform and then describe how the tool works. In the last section, we show how the tool is evolving.

2. JXTA

P2PConference has been developed using the Java implementation of JXTA [10], a network programming and computing platform for P2P systems. Project JXTA was originally conceived by Sun Microsystems and designed with the participation of a small number of experts from academic institutions and industry. The platform was released as an open source project early in the 2001 to become the standard foundation for P2P systems.

The project had to address some issues that were set as objectives [7]:

- Interoperability. Nowadays there are several P2P systems that, though offering the same services (e.g. file sharing), are incompatible because of the lack of a common infrastructure. This issue is referred to as *danger of fragmentation* [14]. JXTA aims at becoming the missing standard and, hence, it has been proposed to IETF [12].
- Platform independence. No target platform (as both programming language and operative system) has been chosen to develop JXTA, thus to embrace a larger base of developers and final users.
- Ubiquity. JXTA has been designed to be implemented on a wide range of digital devices, from cell phones to servers.

At the highest abstraction level, JXTA is a set of six protocols, each defined by XML-based message exchange:

- Peer Discovery Protocol (PDP)
- Peer Revolver Protocol (PRP)
- Peer Information Protocol (PIP)
- Peer Membership Protocol (PMP)
- Pipe Binding Protocol (PBP)
- Endpoint Routing Protocol (ERP)

JXTA technology is designed to provide a layer on top of which other services and applications are built (see Figure 1). Typical P2P software stacks break down into three layers. The lowest level (referred to as JXTA core) deals with peer establishment, communication management, such as routing. In the middle (JXTA services) the layer provides higher level services, such as indexing, searching and file sharing, built upon the low-level features of the core. At the top is the layer of applications (JXTA applications): any P2P system built using the services beneath.

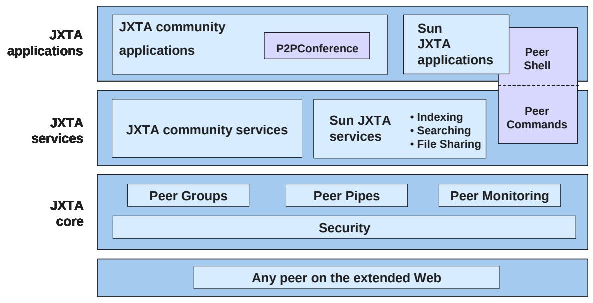


Figure 1. The layered architecture of JXTA

3. P2PConference

P2PConference was inspired by the eWorkshop tool [1] from CeBASE [5]. eWorkshop is a simple web-based collaboration tool to organize and conduct remote, text-based meetings with the aim of gathering and synthesizing knowledge from a group of invited experts. However, P2Pconference is not a mere porting of eWorkshop onto the JXTA platform. Other than replicating the basic features of eWorkshop, we have added new capabilities to run different types of remote conferences, and allow organizers to exercise more control on the participants.

The primary functionality provided by the P2PConference is a closed group chat with agenda, whiteboarding and typing awareness capabilities. The tool allows participants to communicate by typing statements that will appear on all participants' message boards. By responding to statements on the message board, they can carry on a discussion on-line. Around this basic feature, we built other features to help organizers control discussion.

The organization of a remote conference (or simply conference, hereafter) follows a strict protocol which mandates the organizers to choose the main discussion topic, schedule the meeting and decide whether or not to run training sessions (to let participants try out the tool), and, finally, send invitations to participants by e-mail.

Most participants in a conference are experts in their respective domain. Organizing a new conference implies to set up a support team, which consists of the following roles: moderator, director and scribe.

The *director* is the actual conference organizer, since he/she is supposed to choose the main discussion topic and the items that it is composed of, schedule the conference and send invitation e-mails, which contain an user id and password to join the discussion.

The *moderator* is responsible for monitoring and focusing the discussion (e.g. proposing items on which to vote) and maintaining the agenda. Among the support team members, only the moderator is an active participant in the sense that he contributes actual responses during the meeting. He/she is also responsible for assessing and setting the pace of the discussion, that is, he/she decides when it is time to redirect the discussion onto another item.

As the discussion moves from one item to another, the *scribe* captures and organizes the results displayed on the whiteboard area of the screen. When the participants have reached a consensus on a particular item through a vote, the scribe summarizes and updates the whiteboard to reflect the outcome. The content of the whiteboard becomes the first draft of the meeting minutes.

The tool screen has five main areas: agenda, input panel, message board, whiteboard, and presence panel (see Figure 2).

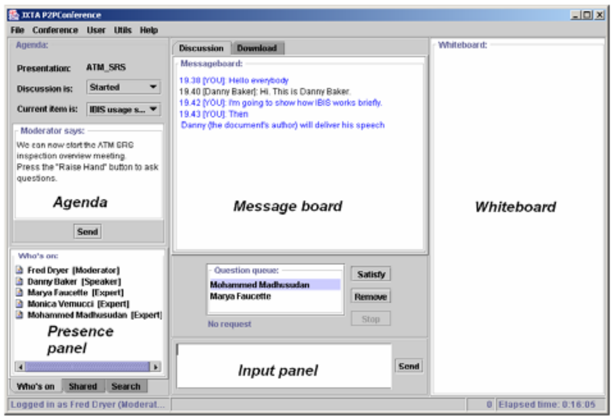
The *agenda* is managed by the moderator and indicates the status of the meeting ("started", "stopped") as well as the current item under discussion.

The *input panel* enables participants to type and send statements during the discussion.

The *message board* is the area where the meeting discussion takes place. Statements are displayed sequentially, tagged with the time of when they were sent and the sender's name.

The *whiteboard* is used to synthesize a summary of the discussion and is controlled by the scribe. In order to realize the goal of measuring the level of consensus among the participants, all of the items added to the whiteboard are subject to voting announced by the moderator. When participants do not agree with how the statements on the whiteboard were formulated, negotiations initiate in order to come up with a more accurate description of the results of the discussion.

The *presence panel* shows participants currently logged in and the played role.



Presentation: ATM_SRS
Discussion is: Started
Current item is: IBIS usage s...

19.38 [YOU]: Hello everybody
19.40 [Danny Baker]: Hi. This is Danny Baker.
19.42 [YOU]: I'm going to show how IBIS works briefly.
19.43 [YOU]: Then
Danny (the document's author) will deliver his speech

We can now start the ATM SRS inspection overview meeting.
Press the "Raise Hand" button to ask questions.

Fred Dryer [Moderator]
Danny Baker [Speaker]
Marya Faucette [Expert]
Monica Vemucci [Expert]
Mohammed Madhusudan [Expert]

Mohammed Madhusudan
Marya Faucette

No request

Logged in as Fred Dryer (Moderat...
0 Elapsed time: 0:16:05

Figure 2. P2PConference screenshot

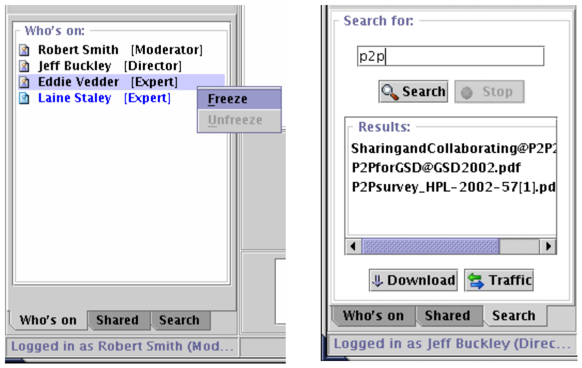
All of these features can also be found in the eWorkshop tool. We further enhanced support for remote conferencing by adding the following features:

- Control. Conference organizers need more control power over participants. Hence, we also added *freezing* − moderator can freeze those experts who disturb, forbidding them to type and ensuring the discussion to flow smoothly (see Figure 3a) − and *hand raising*, that is participants must ask the moderator the right to talk or ask questions.
- File sharing. A collaborative tool cannot be such without file sharing capability (see Figure 3b).
- Protection. A conference is said to be "protected" if it does not allow users to access the drafts (i.e. the discussion log and the whiteboard content) saved by the peer into HTML files. The only participant allowed is the director. This option ensures the organizers that no one else can carry on a conference analysis.

Indeed, the presence of the moderator only prevents the discussion to become unconstrained, ensuring that all of the items in the conference agenda are discussed. This kind of remote meeting is apt for brainstorming sessions with limited or no control over the participants for the organizers. We did not want to bind the organizers to run only brainstorms and, hence, we identified three different types of existing conferences to model and implement in P2PConference:

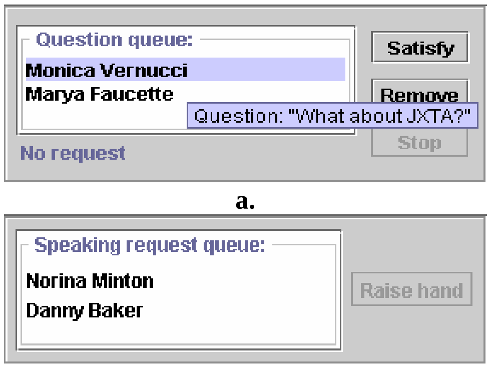
- Meeting. It ensures a limited control power since the moderator can only "freeze" disturbing participants (i.e., the moderator may forbid them to type and send statements). This conference type models simple, remote brainstorms.

- Presentation. This is a more complex kind of conference: one special invited expert, the *speaker*, delivers his own speech and the other invited experts (i.e., the audience) can ask him/her questions, after “raising their hands”. The moderator manages the queue of the asked questions (see Figure 4a).
- Panel. It is a generalization of presentation: there is more than one speaker, the so-called *panelists*, and, since any of them can deliver a speech, they have to request the right to speak by “raising their hands” (see Figure 4b). Moreover, the experts who want to ask a question are to pick the panelist(s) and raise their hands too. Hence, the moderator manages two separate queues, one for the panelists and one for the experts



a. **b.**

Figure 3. The presence panel with freezing menu (a) and the search panel (b)



b.

Figure 4. Hand raising panels for question requests (a) and speaking requests (b)

5. Conclusions and Further Work

In the field of collaborative software development (CSD) environments P2P technology and decentralization have begun to being introduced [2, 3].

In this paper we have described P2PConference, a tool for running remote conferences. The tool is also an open-source software hosted at the Project JXTA site [11]. Currently, one of the authors has the role of project owner, two fifth-year computer science students act as developers (committers), and thirteen people are contributors (mainly for issue reporting and bug fixing).

Much of the tool functionality has been implemented in the first release. Also, we plugged P2Pconference into IBIS [13], a tool developed at the University of Bari to support software inspections for geographically dispersed teams. Using Java Web Start [15], inspectors can launch P2PConference and run a kickoff meeting to provide background information on the inspection process or the product being inspected.

Current work is aimed to make deployment easier, by automating the initial peer configuration, and add support for presentation sharing and co-browsing. As further work, we are planning to develop a remote-conferencing plugin to integrate our tool in an extensible IDE, such as the Eclipse Platform [6].

6. References

[1] V. Basili *et al.*, “Building an Experience Base for Software Engineering: A report on the first CeBASE eWorkshop”, *Proc. of International Conference on Product Focused Software Process Improvement* (PROFES 2001), Kaiserslautern, Germany, September 2001, pp 110-125.
[2] bitkeeper.com, *BitKeeper Source Management*, http://www.bitkeeper.com/Products.BK_Pro.html
[3] S. Bowen, and F. Maurer, “Using peer-to-peer technology to support global software development – some initial thoughts”, *Proc. of the Int. Workshop on Global Software Development (ICSE 2002)*, Orlando, FL, USA, May 2002.
[4] G. Canfora, F. Lanubile, and T. Mallardo, “Can Collaborative Software Development Benefit from Synchronous Groupware Functions?”, *Proc. of the 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes* (CSSE 2003), Benevento, Italy, March 2003.
[5] CeBASE Web Site, http://www.cebase.org
[6] eclipse.org, Eclipse Foundation website, http://www.eclipse.org

[7] L. Gong, “JXTA: A network programming environment”. *IEEE Internet Computing*, 5(3):88--95, May-June 2001.
[8] J. D. Herbsleb and R. E. Grinter, “Architecture, Coordination, and Distance: Conway’s Law and Beyond”, *IEEE Software*, Vol. 16, No. 5, September/October 2001, pp. 16-20, pp.63-70.
[9] J. D. Herbsleb and D. Moitra, Global Software Development, IEEE Software, Vol. 18, No. 2, March/April 2001, pp. 16-20. Software Engineering, *IEEE Software*, Vol. 19, No. 3, May/June 2002, pp. 26-38.
[10] jxta.org, *Project JXTA Home Page*, http://www.jxta.org
[11] jxta.org, *P2PConference Home Page*, http://p2pconference.jxta.org
[12] jxta.org, *IETF standardization effort*, http://www.jxta.org/IETFStandard.html
[13] F. Lanubile, and T. Mallardo, “Tool Support for Distributed Inspection”, *Proc. of International Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, UK, 2002.
[14] D.S. Milojicic *et al*. “Peer-to-Peer Computing”, HP Laboratories Palo Alto, March 2002
[15] sun.com, *Java Web Start Technology*, http://java.sun.com/products/javawebstart

Test-Driven Global Software Development

Bikram Sengupta Vibha Sinha Satish Chandra
IBM India Research Laboratory,
Block 1, Indian Institute of Technology, New Delhi 110016, India

Sharath Sampath K. Guru Prasad
IBM Global Services India Pvt. Ltd.,
Embassy Golf Links Level-3, Bangalore 560071, India
{bsengupt,vibha.sinha,satishchandra,ssampath,guruprasad}@in.ibm.com

Abstract

In a global software development project, distributed teams need to have a consistent view of the system even in the face of frequently changing requirements. Thus how precisely requirements and changes therein are communicated to remote developers becomes a critical issue. In this position paper, we hypothesize that a test-driven methodology may help keep development across multiple sites consistent with changing requirements and with each other.

1 Introduction

According to the iterative model of software development, a project cycle commences with requirements gathering, followed by design, coding and testing; then the next cycle begins. An underlying assumption of this view is that once requirements are collected, they are generally stable through the rest of the cycle. However, the real-world scenario differs considerably. Business cycles are shrinking so rapidly these days that the boundaries between the phases are getting blurred. Very often, by the time developers begin coding, customer requirements are already changing. However, constantly having to change code to meet new requirements is only one half of the problem; in a large project spread across several development teams, a greater challenge lies in ensuring that even as requirements change, a consistent view of the system is maintained across all the teams. Consider for example, a major enhancement request that potentially affects the behavior of many modules; developers then need to clearly understand not only what changes to make in their own module, but also how the behavior of surrounding modules may change. In some cases, the interface agreements may need to be modified; in more subtle cases, there can be behavioral changes in those modules, without any externally visible syntactic changes. Unless there is a shared understanding about these changes, the system may easily slip into an inconsistent state.

In a single-site project, developers usually rely on extensive interactions to clarify doubts regarding requirements and their impact on the behavior of various modules. When the development environment becomes distributed however, face-to-face meetings, if any at all, are few and far in between. There are e-mails, tele-conferences etc. no doubt, but there is a practical limit to their efficacy when it comes to developing a common understanding of the imprecise, ever-changing textual documents that ususally convey requirements. The physical distance and the differences in time-zones make multi-site operation inherently disconnected in nature. Add to this the differences in cultures, languages etc., and it is easy to see why semantic information often do not get across uniformly to remote sites. The resulting discrepancies in understanding introduces delay [5] and may necessitate substantial re-work during integration. Hence there is a need to complement the usual forms of cross-site communication, with practical methodologies that can convey information (in requirements/interface agreements etc.) easily and precisely.

In this position paper, we propose a test-driven methodology to address some of the above challenges in global software development. The basic idea is to create test suites of the different modules prior to development, share them across all the sites and use them as a medium of communication between development teams. For example, changes in these test suites may be used to reflect changes in requirements or in module behavior. The shared understanding that would result from this should help preserve overall consistency.

2 Background

In this section, we (i) briefly discuss some existing approaches to precise behavioral descriptions and (ii) review the notion of test driven development (TDD). We then describe how TDD can be viewed as another precise, although incomplete, form of behavioral specification.

Precise Behavioral Specifications: The software industry has long felt the need for unambiguous specification techniques that developers can use. A number of formal notations have thus been proposed over the years to bring clarity and rigor to software development. These range from mathematical formulations given as algebraic axioms [8] to illustrate the behavior of class methods, to more accessible languages like Eiffel [2], Java Modeling Language (JML) [4], Object Constraint Language (OCL) [6] etc. that are based on the Design By Contract paradigm [1], and use method pre-conditions, post-conditions and class invariants to succintly represent behavior. However, the acceptance of these approaches in the industry has been low in general. Developers are usually unwilling to learn one language for implementation, and another for specification. The complexity associated with these methods may also serve to discourage users and moreover, their technical rigor is often considered an overkill. Finally, they usually do not scale up, and their application in industry-sized projects may be simply infeasible. This points to the need for lightweight but precise specification methodologies that may be used to convey semantic information to developers, and which may be easily integrated with their existing practices.

Test Driven Development: The idea behind Test Driven Development [7] or TDD is simple: before implementing a new functionality, first write executable unit test cases for it. Once you have written enough test cases to thoroughly check the new feature, write the actual code to pass these test cases. The test cases thus become a mini-specification of the functionality that was implemented. This then goes on iteratively as more and more functionalities are added. At the end, one not only has the complete implementation, but also an efficient regression test bed capturing all the new functionalities that were added, and which can be used to identify if subsequent changes break anything in the existing system.

Test Suites as Behavioral Specification: In effect, TDD is a novel approach of creating incomplete but precise specifications on-the-fly during development. Developers have an implicit understanding of what a program is supposed to do, and although they may not be able to specify this behavior formally (e.g. with JML like pre- and post conditions or algebraic axioms), their understanding reflects in the test suites they design. The test suites written prior to development may thus be looked upon as a lightweight specification that guides subsequent implementation. They are obviously incomplete in a formal sense with respect to the full specification, but have several practical advantages: creating test cases requires no new skills from developers, and the specification may be incrementally enriched by adding new test cases as needed. Finally, test suites are unambiguous; from the prespective of global development, this means that a test suite should be interpreted in the same way by different remote development teams.

3 Test-Driven Global Development

We now propose an approach whereby test suites are used as a knowledge sharing medium between remote sites in a distributed development environment. The idea is inspired by the TDD paradigm described above; just as unit test cases written prior to development specify the functionality to be implemented, we believe that early availability of module-wise test suites can serve as a precise documentation of requirements and of module behavior.

Early Test Suites In a typical software development endeavor, once requirements are formulated, some intermediate steps (e.g. use-case diagrams or scenarios) lead to a high level design (modules, interfaces etc.). In a multi-site environment, the modules (e.g. clusters of classes) are then distributed across the remote sites for implementation. The high-level design may be followed by a more detailed design at the remote sites, followed by implementation. Then testing begins, starting with unit testing, to class and module testing, to module integration testing, and finally system level testing.

To adopt TDD in the multi-site context, we propose a simple change to the process outlined above, by suggesting that the end-products of high-level design should not only be modules and interfaces, but also some module-wise test suites jointly created by the system architects and the testing team. These test suites can include unit test cases that illustrate both the normal and exceptional behavior of the public methods, as also functional test cases (e.g. sequences of method calls) that can capture the way a client may use a module. The test cases are associated with the interface, and as such become a first class entity in the design space. At the same time, the interface, usually only syntactic in nature, becomes enriched semantically.

These test cases need not necessarily be fully executable code, but should be precise enough to document the important details e.g. the various input events, corresponding outputs, error-conditions etc. in a proper format. Such artifacts arise naturally as part of testing activities, and executable test cases can subsequently be derived out of them. It may be noted here that, in practice, testing activities sometimes do start before development e.g. test plan documents are often created at the end of high-level design. These activities generally proceed in parallel to development, without contributing to the development effort as such, till the testing phase begins. We feel, however, that test cases may also be looked upon as detailed specification entities, and if these are created upfront and made available to developers, then we can fill a gap between higher level requirements and code. In a sense, test cases are the most precise form of requirements, and their usefulness to developers in deriving requirements understanding has been noted by several practitioners e.g. [3]. Developers work at the code and test case level, so higher level requirements make more sense to a developer if communicated through a medium he/she is familiar with. Hence we also propose mapping the higher-level requirements to these test cases, before coding starts.

Communication through Test Suites To keep distributed development teams in sync with requirements, we next propose that requirements cannot be updated without updates to the associated test cases. Thus for example:

- If a new requirement is added, create test cases for it, and map the requirement to the test cases
- If a requirement is deleted, remove the test cases that have become obsolete
- If a requirement is modified, modify related test cases to clearly reflect this change

These actions have to take place *before* any modification is made to the code. In essence, we change the usual traceability graph originating from the requirements and proceeding through code towards test cases, by having test cases precede source code files instead of following them. Thus, during impact analysis following a requirements change, the test suites have to be updated first. Then these modifications in the test suites guide the developers in changing the source code files.

Another advantage of early module-wise test suites may be in illustrating the behavior of a module, say M, to remote developers who need to use M's functionality. In a distributed environment, the development of the different modules proceeds simultaneously and till M becomes available, a remote developer who needs to use M would write a dummy functionality based on M's interface. Interfaces are primarily syntactic in nature, and are not a rich source of information for someone who wants to use the associated module. However, if we have interfaces enriched with test cases, then a developer, looking at the interface of a remote class, would get a much more clear idea of its behavior and how to use it. This may enable better simulation of M at a remote site, and thus smoothen subsequent integration testing. Moreover, when M's behavior needs to be modified in response to changing requirements, this may once again be conveyed to developers of related modules through changes in the interface test suite. Since test suites are much more precise than text documents, and since they may be made available through a central repository, the need for extensive cross-site communication should decrease, allowing the sites to operate in a relatively disconnected manner.

Our proposal does not, in any way, seek to reduce the importance of conventional post-development testing activities. These should be performed following well-established testing principles, as always. Rigorous testing would definitely require more test cases than can be made available in an early test suite, We believe, however, that if we use test cases only for post-development testing at each site, we make use of their power to validate an implementation, but do not utilize their expressive power. By creating some early test suites, we can not only use them during subsequent post-development testing, but also to convey precise semantic information during actual development.

4 Future Work

We are currently investigating what kind of tool support would be necessary to adopt some of the ideas described above in practice. For future work, we would like to define appropriate metrics to empirically determine the effectiveness of our method in improving multi-site software development.

References

[1] B.Meyer. Design by contract. *Advances in Object-Oriented Software Engineering*, 1991.
[2] B.Meyer. Eiffel: The language. 1991.
[3] E.M.Maximilien and L.Williams. Assessing test-driven development at IBM. *25th International Conference on Software Engineering*, pages 564–569, 2003.
[4] G.T.Leavens, A.L.Baker, and C.Ruby. Jml: A notation for detailed design. *Behaviroal Specifications of Businesses and Systems*, pages Chapter 12, 175 – 188, 1991.
[5] J.D.Herbsleb, A.Mockus, T.A.Finholt, and R.E.Grinter. Distance, dependencies and delay in a global collaboration. *ACM Conference on Computer Supported Collaborative Work*, pages 319–328, 2000.
[6] J.Warmer and A.Kleppe. The object constraint language, precise modeling with UML. 1999.
[7] K.Beck. Test driven development: By example. 2002.
[8] R.Doong and P.Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodologies*, 1994.

Using Iterative and Incremental Processes in Global Software Development

Maria Paasivaara and Casper Lassenius
Helsinki University of Technology
Software Business and Engineering Institute
POB 9210, FIN-02015 HUT, FINLAND
Maria.Paasivaara@hut.fi, Casper.Lassenius@hut.fi

Abstract

Iterative and incremental development seems to be a viable approach providing several benefits in inter-organizational distributed software development. This paper presents initial results from an interview study on the usage of iterative and incremental development in inter-organizational distributed software development projects. We describe identified practices, such as delivery synchronization, design and code reviews, communication emphasis, feature-based development, behavioral patterns, and frequent deliveries. We also present the benefits that the use of these practices brought, such as transparency of progress, increased developer motivation due to rapid feedback, flexibility regarding changes, the possibility to involve subcontractors early, ensuring joint understanding of requirements, and the avoidance of "big bang" integration. It seems that the advantages of using the practices overweigh the extra communication and coordination cost they incur.

1. Introduction

Global inter-organizational software development, including outsourcing, subcontracting and partnerships, is becoming increasingly common. Projects developing genuinely novel products are often faced with uncertainty regarding, e.g., requirements and implementation technologies. However, subcontractors or partners often need to be involved long before these uncertainties can be resolved. In such projects, the parties cannot receive clear requirement specifications at the beginning. Instead, close cooperation and communication between the parties is required during the whole project, as the project both builds a product and tries to understand what to build at the same time. In these kinds of projects, problems often arise, since practices and processes needed for collaborating across distances and organizations are neither well understood in theory, nor typically established in practice [9].

For software development facing uncertainties and unpredictable changes, literature suggest the use of iterative and incremental development (IID) as a process model, since it enables fast reaction to changes [6]. IID means that the system is grown via iterations, incrementally adding new features [6]. Global software development literature contains some reports of good experiences of using IID also in distributed settings (e.g. [1, 2]). However, these studies do not report in detail how IID should be implemented and used successfully in distributed projects nor what benefits and drawbacks its use brings.

In our interview study concentrating on collaboration practices in globally distributed projects, we noticed that IID was used in many of the projects studied and that these projects had gained several advantages from its usage [9]. It seems that IID suits distributed development extremely well and helps reduce problems caused by distribution. However, IID also requires close collaboration and communication, which can be hard to achieve in distributed development.

In this paper we report experiences of using IID in five globally distributed inter-organizational software development projects. We present some interesting findings of how companies are using IID in their distributed projects and what kind of benefits they have gained. Since our larger study concentrated on all collaboration practices used in these projects, we could not yet go very deep into IID related practices. The purpose of this paper is therefore to give an overview of our findings related to the usage of IID in distributed development, as well as to motivate further research into its use, benefits and drawbacks.

The rest of this paper is structured in the following way: The next section briefly presents related literature. After that we describe the research methodology and introduce the case companies and projects studied. In the results section we present the experiences we collected from our case projects. Finally, we present a short discussion of the results and their managerial implications, as well as give ideas for future work.

2. Related Work

Global software development literature lists many challenges related to distributed development, e.g., interdependencies among work items that are distributed, difficulties of coordination, difficulties of dividing the work into modules that could be assigned to different locations, conflicting implicit assumptions that are not noticed as fast as in collocated work, and communication challenges [8]. Literature suggests dividing the work into separate modules that can then be distributed to different sites to be developed [4]. These modules should be so independent that communication between sites can be minimized [4]. The authors emphasize that it is possible to split only well-understood products where architecture and plans are likely to be stable. However, in a development environment with a lot of uncertainties, dividing the software into modules and specifying the modules in detail up front is often impossible. Moreover, first specifying and dividing work and subsequently integrating all in “a big bang” is challenging, since integration can cause huge unexpected problems. As a solution Battin et al. [1] suggest an incremental integration plan, which is based on clusters and shared incremental milestones to avoid “big bang” integration. This strategy was used successfully at Motorola. Ebert and De Neve [2] provide similar experiences on the usage of incremental development at Alcatel, where they developed each increment within one dedicated team and based their progress tracking on successfully integrated and tested customer requirements. The authors report that a stable build proved to be one of the key success factors and that the globally applied continuous build improved the project’s cycle time. Neither Alcatel nor Motorola report their integration intervals, but it seems that they did not use very frequent regular build cycles, such as daily or weekly builds. However, even very frequent builds are possible in distributed development. Karlsson et al. [5] report using daily builds and feature-based development successfully in distributed projects at Ericsson.

IID is a core practice in agile methodologies for collocated projects [6, 7], but its use in distributed development has not yet received much attention. Fowler [3] and Simons [11] recently reported their experiences on using agile methods in offshore software development projects. According to Simons [11], an iterative model seems to work well in distributed projects and can eliminate some of the problems that distribution brings. Continuous integration and build verification tests solve integration problems in small steps and avoid large integration problems at the end of the project. Moreover, IID provides increased visibility into project status, which makes it easier for project managers and customers to follow project progress [11]. Fowler [3] discusses the suitable iteration lengths for offshore projects and concludes that iterations cannot be shorter than two weeks, because of the communication overheads of distributed development, and two to three month iterations can already be too long.

This short literature review shows that IID seems to be a viable process model for distributed software development projects, providing several advantages. However, the reported experiences of its use in distributed environment are still quite limited. We believe that collecting more real-life experiences of the usage of IID and the gained advantages would be helpful to managers designing their distributed projects. In this paper we report our initial research results, discuss the benefits of IID in distributed inter-organizational development, and outline ideas for future research topics.

2.1. Research Methodology

The research presented in this paper follows the case-study approach [12] and is a part of a larger multiple case study [9]. The aim of the larger study was to collect successful collaboration practices from inter-organizational software development projects. We used purposeful sampling [10] and selected ten projects from eight companies that we knew used software subcontractors and that we expected to be experienced in inter-organizational software development. We selected projects that demanded constant collaboration and lots of communication between the parties, e.g., due to a high degree of uncertainty, dependencies and changing requirements.

One of the successful practices we found in the larger study was the use of iterative and incremental development. From the ten projects we studied, five used an IID model. In this paper we report experiences collected from those five projects. In these five projects we performed 29 semi-structured interviews, each lasting 2–3 hours.

After our first interview round and data analysis we noticed that IID was a central theme in these five projects. We also noticed that project A was the most interesting project in this sample regarding IID. Therefore we chose to do one extra interview with a manager from that project concentrating only on experiences of IID. We had interviewed this person also during our first interview round and wanted to ask more detailed questions on IID. Basic information about the projects and the number of interviewees is shown in Table 1. The next paragraphs provide short descriptions of the studied projects.

Project A was a new product development project with lots of uncertainty concerning requirements and technology. The German office of a Finnish customer company did this project with the help of two new subcontractors, one from Germany and one from Ireland.

Project B was from the same Finnish customer company as project A. This project also contained a lot of uncertainty

Table 1. Case project interviews

Case projects		A	B	C	D	E
# of interviews	Partnership manager	1	2	1	-	1
	Process developer	1	1	-	-	1
	Project manager	1	2	3	2	3
	Team member	-	-	2	-	1
	Sub-contractors	-	1	5	-	1
	All	4	5	11	2	7
Industry		Tele-com	Tele-com	Fi-nance	Inter-net SW	Be-spoke SW
Distribution (# of sites)		Eu-rope (3)	Eu-rope (4) & North Amer-ica (1)	Eu-rope (3)	Eu-rope (1) & Asia (2)	Eu-rope (6)

regarding requirements and technology. Projects A and B were both subprojects of larger product programs. Project B used a Finnish subcontractor with sites in Finland and Hungary. The customer company had sites involved both in Finland and in the US.

Project C was a a large new product development project done by a Finnish company. Additional resources were assigned from its newly acquired subsidiaries in Denmark and Switzerland.

Project D was a development project for a customer specific system carried out by a small Finnish company, which had its sales and project management in Finland. All development work was performed in a partly owned subsidiary in India.

Project E developed a well-defined customer specific system and was distributed to six sites. In addition to the Finnish customer company's three own sites, also a subsidiary in Estonia and a subcontractor with two sites in Finland were involved.

3. Results

In this section we first introduce our findings on how companies were using IID in distributed settings. The companies used practices such as synchronization of deliveries, design and code reviews, emphasis on communication, feature-based development, behavioral patterns, and frequent deliveries. Then, we present some of the benefits of using IID, such as transparency of progress, increased developer motivation due to instant feedback, flexibility to do changes and start early with subcontractors, ensuring understanding of requirements, and the avoidance of big bang integration. To summarize, our results seem to indicate that IID is a suitable approach for distributed projects and that more detailed studies on the methods and practices of using it are needed.

3.1. Identified Practices

Delivery synchronization In an inter-organizationally distributed project different partners might have different delivery and integration cycles, but our case studies show that synchronizing the delivery and integration cycles between participants is beneficial.

In project A the original plan was to use the waterfall model, but after some quality and schedule problems the customer company and the Irish subcontractor started to use an iterative development model with weekly builds. However, getting used to this new weekly rhythm was not easy. Especially coordinating the work between different teams and specifying the work well enough required learning. The customer tested each build once a week for one day and after that everybody got this tested build as a new baseline. During the early phases of iterative development the teams learned that it was better to develop only small additions at a time to avoid problems. The German subcontractor delivered in longer intervals, only once in 1-3 months. This caused additional work in the integration phase, because the amount of new code was large and not always compatible with the baseline. Finding bugs from 1–3 months worth of work was not easy. In this project the baseline was available to everyone through a common repository or its replica. This made it possible for all partners to test their new code against the baseline before integration.

Project B faced problems with delivery cycles of different length, since two sites used one-week iterations, and one site had a two-month cycle. This led to problems for the subcontrator, since it ended up waiting up to two months for fixes from the customer site using long iterations.

Our interviewees from projects A and B emphasized that when using IID it is important that all participants synchronize the iteration cycles, i.e., use the same length for iteration cycles and deliveries. If this is not done, problems will occur.

The iteration and delivery cycles used varied between projects and also between project phases, e.g., in the beginning they could be longer and in later intensive phases they were shorter. In project B, early phases used three month increments, and the late phases weekly deliveries.

Design and code reviews Design and code reviews seemed to be useful in distributed projects with distant sites or subcontractors. These reviews are early checks that the distributed teams have understood the requirements correctly and are doing what they are supposed to do. In later stages, the deliveries of code fulfill this need. The distributed sites also felt that these reviews were very useful since they got immediate feedback on their work.

In project C iterations were used only with the company's Swiss subsidiary. Their work consisted of three months work with two planned iterations. This project was the first collaboration effort after the Swiss subsidiary had been bought, therefore starting the project required similar efforts as with subcontractors. Before the coding became intensive the customer's Finnish contact person visited the Swiss team twice, first having a design review and then a code review. After that the implementation could safely start and everybody knew that the work was on the right track.

Also in project E code reviews were used in the early phases of the project with the subcontractor and the foreign subsidiary. These reviews were very much appreciated since developers got immediate feedback on their work.

Emphasis on communication Communication requirements in distributed projects using IID and therefore collaborating closely are very high. Especially the projects that had weekly integration cycles, A and B, found communication as a very important prerequisite to be able to work that fast.

Project A had weekly integration meetings, where integration related problems were discussed. These meetings made it possible to learn from mistakes already in the early phases of development. Project progress monitoring also took place during these meetings: only tasks that had passed the tests and been integrated into the build were regarded as ready. Subcontractors could not participate in these meetings, because of security issues concerning this totally new product, but the customer had project managers that represented each of the subcontractors in the meetings and delivered information to the subcontractors. In this project only the Irish subcontractor participated in the weekly cycle. Frequent communication with this subcontractor was ensured by having their staff sitting at the customer's premises. Ad-hoc communication and meetings were encouraged in this project. Also the "behavioral patterns" used in project A, and described later on, are closely related to emphasizing fast communication and getting answers quickly.

Project B had a normal weekly face-to-face meeting in all its teams. The following day project managers both from the customer company and the subcontractor had a weekly teleconference. The subcontractor's team leaders could also participate in this meeting if they deemed it necessary; otherwise they could read the meeting memos.

Both projects A and B had higher-level monthly meetings. In project A they were called R&D meetings. In Project B they were project steering group meetings, which were organized every time at different project sites to enable both managers and developers from the different sites to meet face-to-face.

Project C, having only two iterations, also found frequent communication to be an important factor for the project's success. The Finnish customer company had one person responsible for all communication with its Swiss subsidiary. This person felt that his task was to answer all questions as soon as possible. These fast answers were very much appreciated by the Swiss developers. Moreover, this contact person had three collocated stays with the team in Switzerland, each lasting about one week. This, of course, facilitated communication and built trust, which was also regarded important.

Project D delivered a customer specific system using IID. The main contractor was a Finnish company that used its partly owned Indian subcontractor as a development resource. The Finnish office negotiated the requirements with the customer, made a requirements specification document and delivered it to India. The subcontractor's project manager commented on the requirements by email and asked detailed questions. The Finnish project manager answered the questions by email and discussed difficult issues through chat. The aim was not to create a perfect specification, since the project's customer could not provide that. Instead, the project was specified to such a level of detail that the Indian subcontractor could develop an initial version of the system. After the delivery of this initial version the Finnish main contractor commented on it. And then, after some improvements, also the Finnish customer commented on the system. The project had several of these comment-improvement rounds. During the whole development, the Indian developers were encouraged to ask questions through chat from the Finnish main contractor's project manager. The customer was also able to monitor project progress by reviewing the code that the Indian developers checked in to a repository located in Finland several times a day. This well functioning communication process was used in all projects between this main contractor and its Indian subcontractors.

Feature-based development Project A had clearly separate sub-areas that could be given to each of the subcontractors. Because builds were weekly and the customer wanted to do functionality testing, the work had to be coordinated quite tightly so that all code affecting certain functionality would be ready at the same time. This feature-based development meant that small increments done in different modules had to be evolving in good synchronicity, in order to enable proper testing and to avoid difficult merging of code later on. In Project A, the customer's project manager made a monthly plan of the tasks to be performed, and the subcontractors' project managers made weekly plans of their internal tasks.

Behavioral patterns Project A had noticed that using an IID model in distributed development did not automatically bring all necessary practices needed for successful cooperation. This project developed additional practices they called "behavioral patterns" to complement the development model. These practices cannot easily be described in process descriptions but are very essential to IID according to our interviewee from project A. He ensured that in cooperation with subcontractors and partners these practices have been equally valid and important as in internal development. According to him, project A had 16 behavioral patterns, that were developed during the project concerning, e.g., management, personal behavior, and the use of tools and work processes. Our interviewee presented three examples of the practices that, based on his experience, were important for the success of their very short-cycled IID process.

A practice called "immediate escalation of issues", means that problems have to be brought up right away. The "project manager always available" practice is closely related to the previous one, meaning that when a developer gets stuck he can immediately ask for help from the project manager who has to be available. "Immediate decisions" means that decisions have to be made fast and not left to later meetings, so that work can continue. This last practice was felt to be more difficult to use across distances when people making decisions might never have met face-to-face and this can easily lengthen decision-making.

Frequent deliveries Project E used frequent deliveries when designing and implementing a large customer specific system. The requirements were quite stable and well-known. The customer company divided the work into small tasks and specified, e.g., all windows and services in detail. These well-specified tasks were then given to subcontractors and internal sites for implementation. Specification work and coding took place at the same time. Both a subcontractor company and an own subsidiary received tasks for 2–3 weeks at the time. When the tasks were done, a delivery was made, and new tasks were assigned. The problem with this way of working was that these delivered services and windows had dependencies to other services or windows, and the customer company could test them only after all related functionality was ready. Therefore, getting test results could sometimes take as long as half a year after code delivery. Clearly, this project would have benefited from better synchronization of deliveries.

3.2. Benefits gained

Transparency of progress Frequent delivery cycles and integration brought transparency of work progress to all partners. When both the customer and the subcontractor used IID, the subcontractor regularly delivered functioning code during the development phase, e.g., monthly or even weekly. Our interviewees told that when deliveries were integrated and tested right away this gave a very good picture of how the project was progressing. They had noted that frequent deliveries made it easier for the customer to monitor the real progress of the subcontractor's work.

Instant feedback Integration and testing reports gave distributed developers instant feedback on their work, which they felt was very motivating. Moreover, when the customer saw that the subcontractor was doing a good work, the customer's personnel started to trust and respect the subcontractor and its developers' know-how, which made further collaboration easier.

Flexibility From the customer's point of view IID brings additional flexibility, when the customer can do changes also during the development phase without time consuming negotiations with subcontractors. Of course, a suitable type of contracting has to be chosen. IID also enables the customer company to take subcontractors into the project already in the early phases of development, when requirements cannot yet be specified in detail. With this kind of development process it is no longer necessary to specify all requirements before subcontractors are involved; instead, since requirements are allowed to change during the project, work can start despite technological or goal-related uncertainties. However, this requires all parties to have "an experimental mindset" and fast and open communication with each others.

Ensuring joint understanding of requirements In IID frequent integration and testing ensured that the subcontractor had understood the requirements correctly. This is a typical uncertainty in distributed development, especially when companies have not worked together before and have different cultures. Frequent integration and testing gives fast feedback and any misinterpretations become visible early. Thus, possible misunderstandings have less damaging consequences. Moreover, learning from mistakes is fast and happens early, preventing problems from accumulating and creating situations that are harder to resolve.

Avoiding "big bang" integration IID,as used in our case companies, prevented different sites and partners from doing too long periods of independent development, which could have led to modules that would be hard or impossible to integrate, i.e., they avoided possible problems that would come from "a big bang integration".

4. Discussion and Conclusions

Frequent communication is a central prerequisite for successful implementation of IID in a distributed environment. This way of development requires much more communication between parties during the whole collaboration compared to development where work products can be clearly separated into independently developed modules. Especially uncertainties in the form of changing requirements demand communication and problem solving. In uncertain environments short iteration cycles are needed to reveal problems as early as possible. The shorter the cycle the more communication is needed to coordinate the work and to quickly solve the problems during development. When the cycles are longer the communication need is more concentrated to the integration phase. This communication overhead is clearly an issue that requires careful planning when implementing IID in a distributed environment.

Defining an iteration length that is suitable for distributed development is an interesting topic. In our study both projects A and B, used one week integration cycles successfully also with subcontractors. Fowler [3] reported that iteration cycles should not be shorter than two weeks in offshore projects due to communication overhead. One explanation to this difference could be that in our study all parties that participated in weekly iterations were from Europe and therefore the time-zone difference was quite minimal. Moreover, the subcontractor in project A had on-site personnel at the customer company, which facilitated communication. Fowler, however, worked with projects distributed between India and Europe or North America, where time differences are noteworthy.

4.1. Limitations

Our initial results presented in this paper are based only on a few case projects, which limits our possibility to draw far reaching conclusions. Moreover, when doing our first interview round we did not concentrate on studying IID, but asked about many other practices too. This means that we could not collect as deep a knowledge on IID and related practices that a more focused study could have provided. The selection of our case projects did not concentrate on finding interesting cases just from the point of view of IID.

4.2. Managerial Implications

We think that our results have several implications for managers working with distributed development. First of all, it seems that short increments are suitable for distributed development, especially when the project faces high degrees of uncertainty. However, when using increments, it is important that all partners use the same iteration length.

To enable reasonable testing of functionality, feature-based development using tight control can be used. Also the use of design and code reviews in the beginning especially with new subcontractors helps to ensure that they have understood the coding standards and requirements correctly. The feedback also motivates them. Finally, frequent communication and problem solving is essential in distributed IID. Efficient communication requires both planned communication practices and assigned resources.

4.3. Future Research

In the future we plan to study additional case projects using IID in distributed environments. We think that it is important to get deeper insights on how these challenging projects really work, what kind of practices are used, what the major problems are, for what kinds of projects this model of working is suitable. Another interesting future research topic is tool support for this kind of projects.

References

[1] R. Battin, R. Crocker, J. Kreidler, and K. Subramanian. Leveraging resources in global software development. *IEEE Software*, pages 70–77, March/April 2001.

[2] C. Ebert and P. De Neve. Surviving global software development. *IEEE Software*, 18(2):62–69, 2001.

[3] M. Fowler. Using agile software proces with offshore development. *http://www.martinfowler.com/articles /agileOffshore.html*, 7.1. 2004.

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

[5] E. Karlsson, L. Andersson, and P. Leion. Daily build and feature development in large distributed projects. In *ICSE-2000*, pages 649–658. IEEE Computer Society Press, 2000.

[6] C. Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley, Reading, MA, 2003.

[7] C. Larman and V. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, pages 47–56, June 2003.

[8] A. Mockus and J. Herbsleb. Challenges of global software development. In *7th International Software Metrics Symposium*, pages 182–184. IEEE Computer Society, 2001.

[9] M. Paasivaara. Communication needs, practices and supporting structures in global inter-organizational software development projects. In *ICSE International Workshop on Global Software Development*, Portland, Oregon, 2003. IEEE Computer Society.

[10] M. Q. Patton. *Qualitative evaluation and research methods*. Sage Publications, Newbury Park, Calif., 2nd edition, 1990.

[11] M. Simons. Internationally agile. *InformIT*, 15.3. 2002.

[12] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, Thousand Oaks, CA, USA, 2nd edition, 1994.

Communication Problems in Global Software Development: Spotlight on a New Field of Investigation

Sébastien Cherry, Pierre N. Robillard
Software Engineering Research Laboratory, École Polytechnique de Montréal
{sebastien.cherry, pierre-n.robillard } @ polymtl.ca

Abstract

While it is widely recognized that communication plays a critical role in software development, it has been observed that problems of coordination may be generated when teammates in the field are working at a distance from one another.

This paper presents an ongoing empirical study on ad hoc collaborative activities which occur in an industrial software engineering environment. We believe that a better understanding of these types of activities and their content will pave the way to further solutions designed to enhance communication, and thus improve both collaboration and coordination in virtual software development settings.

We include details of our motivations for the study, followed by some methodological considerations, and, finally, some preliminary results which demonstrate not only the significance of our data, but also the relevance of our approach.

1. Introduction

Communication is undoubtedly an essential element which plays a critical role in a software engineering process in the gathering and crystallization [15] of all relevant information in quality software which fulfills the user needs on time and within budget [2], [3], [13], [14], [16]. Moreover, several studies have stressed the fact that informal communications seem to be fairly important in terms of the time spent on a software project. Perry, Staudenmayer and Votta (1994) found during a case study that informal communications take up an average of 75 minutes per day per software developer [12], and Robillard and Robillard established in another case study that ad hoc collaborative activities can occupy up to 41% of the developer's time [14]. Seaman (1996) [16] also supports the need for this type of communication if developers are to carry out their tasks adequately.

However, in the global software development setting, which has become a more common practice for many business reasons, some researchers have observed that communications, specifically informal ones, face significant challenges by virtue of distance, both geographical and temporal [6], [7], [8], [9]. They note that the consequence of this is a potential for problems of coordination to occur.

This short paper presents an ongoing empirical study being carried out within the framework of a case study in the industry which explores the ad hoc collaborative activities that take place in a software engineering setting. By ad hoc collaboration, we mean all informal and spontaneous activities performed by two or more developers who are working on a particular task of the project. These activities can take many forms, such as in informal peer-to-peer conversations, also referred to in the literature as "water-cooler-talk" [6], as well as electronic mail exchanges and phone calls.

We believe that a better understanding of such activities and their content will pave the way for further solutions with the aim of enhancing communication and thus improving both collaboration and coordination in virtual software development settings. Our general approach is to observe and understand the informal and spontaneous collaboration activities that take place in a classical single-site software development environment where developers have as much freedom and opportunity to communicate as they wish, and to measure their positive and negative impacts on the rest of the software project. Depending on the results of our investigation, two avenues of action can be envisaged. The first might be to infer and formalize from our observations some state-of-the-art rules or practices applicable in global software development contexts which will be better adapted to the empirical reality and make collaboration between teammates working apart more efficient. The second might be to use this comprehension to give us some insight into the tools needed to support communications in a distributed software development environment with the intention of creating what some call the "virtual 30 meters" [9].

In this paper, we explain our research methodology in broad strokes and present some preliminary results which demonstrate not only the significance of our data, but also the relevance of our approach.

2. Research Methodology

Empirical studies have been becoming more and more popular in the past few years in software engineering, in parallel with the growing popularity of people-centered researches. Indeed, researchers must innovate in order to study this new topic of interest, namely people, by borrowing certain techniques used in the social sciences, such as psychology and sociology. Our research methodology has been inspired by several papers [4], [11], [18], [19] as well as aforementioned studies which have examined the human aspects of software engineering, but has been to some extent adapted to the field investigated in this study. Below, we give a general description of our research methodology.

2.1. Research Objectives

As stated above, good communication is the *sine qua non* of software development processes in obtaining quality products which meet user needs on time and within budget [2], [3], [13], [14], [16]. Research has shown the non-negligible importance of informal communications [12], [14], [16], while some has specifically highlighted the fact that distance in global software development is a challenge to informal communications which can generate problems of coordination [6], [7], [8], [9]. However, no research has described the content of this type of communication. These elements led us to define the following research objectives:

- To design a model of the ad hoc collaborative activities found in an industrial software engineering setting and characterize them, as well as to identify and describe the content of the communications that ensue.
- To generate a series of hypotheses emerging from the results of this research which could later be validated by confirmatory research.

2.2. General Approach

These objectives will be achieved by means of participant observation. This technique is suitable in exploratory research like ours where the goal is to inductively generate theories from direct observations, also called grounded theory, rather than to empirically verify a hypothesis formulated in advance [1], [10], [17].

2.3. Target-setting

The target is a team of eight individuals which develops software for commercial purposes in a large international company which has been in operation for several years and which has a well-established software development process. It should be noted that, even though the observations are made in a large company, this last contains attributes of small or medium-sized organizations since the work is divided among small teams like the one that is participating in our research. Also, the members of this team are highly representative of developers in the industry, in terms of the wide variety of their ages, software development experience, schooling and length of service in the company.

2.4. Data Collection

The data collection phase, which lasted 8 weeks, is done. The methods used during this phase were selected following an earlier ethnography period of several months. The data collected during this period includes:

- 185 hours of audio-video recordings of working sessions
- 2496 electronic mails exchanged by the 8 teammates
- A daily backup of the source code and other artifacts found in the field

Audio-video recordings were preferred over field notes because they offer the enormous advantage that they can be consulted many times over. This is very important in exploratory research like ours where we do not necessarily know in advance what to observe. E-mails were automatically captured by triggers defined in the messaging software used in the company, and include both those received and those sent to allow cross-validation. Finally, a backup was made of the source code and artifacts found to allow further content analysis.

2.5. Data Analysis

The principal method used to analyze the large amount of data, mainly in the form of the audio-video recordings, was the Exploratory Sequential Data Analysis (ESDA) [5]. This method was chosen because it is particularly well-suited to exploratory research like ours, where theories have to be induced from empirical data, and, more importantly, where the sequential information of the data must be preserved.

Briefly, the ESDA process is iterative. It involves the definition, sometimes intuitive, of concepts arising from the ESDA tradition of taking the point of view of the researcher, subsequently providing a guide as to what to observe in the raw data and how to manipulate it to derive data on which theories will be founded. This process is done iteratively because it is often necessary to step back in order to add, remove or revise some concept definitions [5].

Of the eight different ways to manipulate data proposed by ESDA, encoding is certainly the most important. This consists of labeling each data sequence with a code formed from an exhaustive, exclusive and limited list of categories. This allows qualitative data to be transformed into quantitative data, which in turn makes it possible to further manipulate the data, by performing statistical calculations, for example [5], [17].

3. Preliminary Results

The following results are based on observations made on the activities of four of the eight developers on the team over a period of 8 hours each. It does not take into account e-mail exchanges. Also, the four individuals observed were chosen because they have been seen to collaborate more with their teammates than the others. This choice is justified because our purpose is to study the content of the ad hoc collaborative activities that occurred, so that these particular individuals were simply more likely to give us more data to analyze.

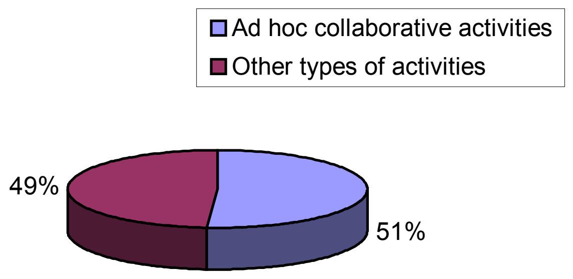


Figure 1. Distribution of time spent in ad hoc collaborative activities in comparison with other types of activities

As Figure 1 illustrates, 51% of the time spent on the project by the observed developers is occupied on ad hoc collaborative activities, in contrast with other types of activities. This is a surprising result which seems to strongly confirm the importance of the phenomenon observed in the target setting, but which needs to be validated by further analysis on a much larger scale.

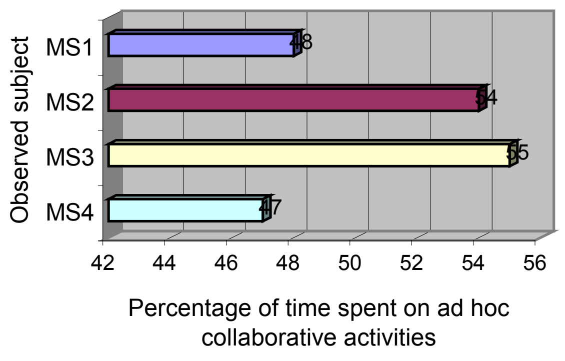


Figure 2. Percentage of time spent on ad hoc collaborative activities by observed subject

Figure 2 shows the time spent on ad hoc collaborative activities per observed subject. It can be noted that the percentages associated with subjects MS2 and MS3 are slightly higher than the others. This difference can possibly be explained by the nature of the work performed by these subjects since MS2 is the team's project manager and MS3 is in charge of the infrastructure for the software built and often the problem-solver on the team. An interesting thing to note is that, in 78% of the interactions in which MS2 is involved, his interlocutors initiated the interactions. However, it has been noted in the field that, most of the time, MS2 shares information by broadcasting a message to his team via e-mail instead of in peer-to-peer conversations. Thus, it will be interesting to analyze these e-mail exchanges. By the way, in each of the 82 interactions observed, an average of 2.3 stakeholders took part, and their average duration was 6:31 minutes.

As for the ad hoc collaboration activities observed, a preliminary outline has emerged from the raw data containing six categories, as follows: "cognitive synchronization" exists when two or more developers exchange information to ensure that they share the same knowledge or the same representation of the object in question; "problem resolution" occurs when two or more developers are aware of the existence of a problem and attempt by various means to solve the problem or to mitigate it; "development" occurs when two or more developers contribute to the development of a new feature or component of the software; "management" is the result of two or more developers coordinating and planning activities such as meetings, common working sessions or scheduling; and "conflict resolution" is the process of two or more developers taking part in discussions to resolve a difference of opinion. Ad hoc collaborative activities in the "not relevant" category, group together all the interactions that do not concern the project or the software built.

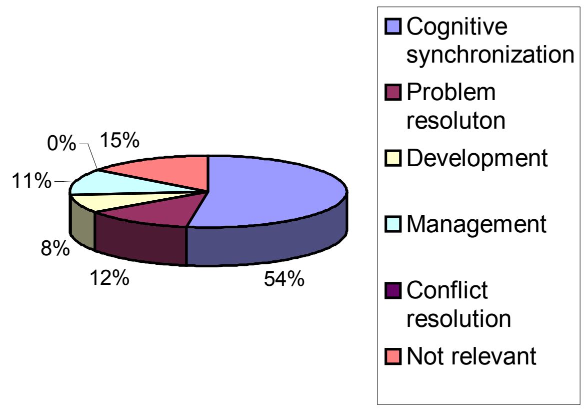


Figure 3. Distribution in number of occurrences of ad hoc collaborative activities identified

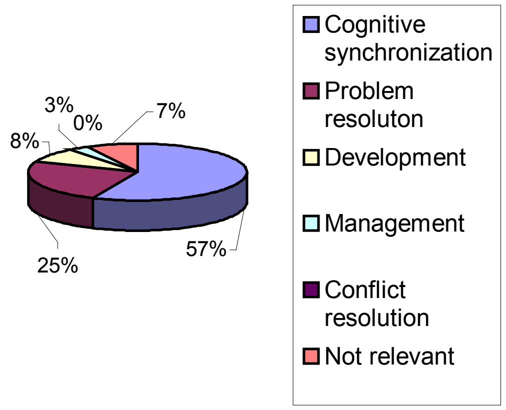


Figure 4. Distribution in terms of time spent on ad hoc collaborative activities identified

Figure 3 shows that, in a little over half the times when these interactions occur, they do so in the form of cognitive synchronization, and this tendency is supported by the data in Figure 4 which show the distribution in time spent. This is not a surprising finding, since it is well established that the sharing of information and knowledge is a crucial element in software development.

Moreover, it can be noted that problem resolution activities represent only 13% of the occurrences, but that, in terms of time spent, the percentage rises to 25%. This suggests that, when they occur, problem resolution activities take much longer than the others. This is supported by the statistics of time spent by sequence as function of ad hoc collaborative activities, which shows that a mean of 9:48 minutes is spent on problem resolution.

Finally, another interesting finding is that management activities, unlike problem resolution, represent 13% of the occurrences, but only 3% in terms of time spent. In other words, they are quite numerous relative to the small percentage of time spent on them. This result probably tends to support the theory of some researchers to the effect that informal communications are important in order that teammates can coordinate their activities efficiently [6], [7], [8], [9].

4. Conclusion

It is widely held that communication is a crucial element in software engineering, but, unfortunately, it is an aspect which seems to be lacking in global software development and one which must be addressed.

This paper presents an ongoing empirical study on ad hoc collaborative activities in an industrial software engineering setting. The general objective of this research is to gain a better understanding of these kinds of activities and their content in order to be able, subsequently, to propose software process enhancements with the aim of rendering collaboration between teammates more effective on the one hand, and, on the other, to obtain some insight into the tools needed to support communications in a distributed software development context.

We think that this kind of research is needed, first of all, because the importance of communication in distributed environments is poorly understood, but also to expose any wrong assumptions there may be that are often mistaken for the truth in software engineering.

Even if further analysis are to be done, the few preliminary results that were partially presented in this contribution tend to demonstrate that a data model and certain patterns are emerging from the vast quantity of data amassed turning the spotlight on a new facet of the empirical reality of software engineering which until today was completely hidden.

5. Acknowledgments

This research would not have been possible without the agreement of the company in which it was conducted, and without the generous participation and patience of the software development team members from whom the data was collected. To all these people, we extend our grateful thanks.

6. References

[1] Babbie, E., *The Practice of Social Research, 9th edition*, International Thomson Publishing Company, 2001.

[2] P. D'Astous, and P.N. Robillard, "Empirical Study of Exchange Patterns during Software Peer Review Meetings", *Information and Software Technology*, 44, 2002, pp. 639-648.

[3] P. D'Astous, P.N. Robillard, F. Détienne and W. Visser, "Quantitative Measurements of the Influence of Participant Roles during Peer Review Meetings", *Empirical Software Engineering*, 6, 2001, pp. 143-159.

[4] Fenton, N.E. and S.L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach*, PWS Publishing Company, 1997.

[5] C. Fisher and P. Sanderson, "Exploratory Sequential Data Analysis: Exploring Continuous Observational Data"*, Interactions*, 3:2, Mar. 1996, pp. 25-34.

[6] R.E. Grinter, J.D. Herbsleb, and D.E. Perry, "The geography of coordination: Dealing with distance in R&D work", *GROUP'99: International Conference on Supporting Group Work*, Coordination and Negotiation, 1999, p. 306-315.

[7] J.D. Herbsleb and R.E. Grinter, "Splitting the Organization and Integrating the Code: Conway's Law Revisited", *Proceedings, International Conference on Software Engineering*, Los Angeles, CA, 1999, pp. 85-95.

[8] J.D. Herbsleb and D. Moitra, "Guest Editors' introduction: Global software development", *IEEE Software*, 18:2, March/April 2001, pp. 16-20.

[9] 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, June 2003, pp. 481-494.

[10] D.L. Jorgensen, *Participant Observation A Methodology for Human Studies*, Applied Social Research Methods Series; v.15, Sage Publications, 1989.

[11] B. Kitchenham and al., "Preliminary guidelines for empirical research in software engineering", *IEEE Transactions on Software Engineering*, 28:8, 2002, pp. 721–734.

[12] D.E. Perry, N. Staudenmayer and L.G. Votta, "People, Organizations, and Process Improvement", *IEEE Software*, July 1994.

[13] P.N. Robillard, "The Role of Knowledge in Software", *Communications of the ACM*, 42:1, 1999, pp. 87-92.

[14] P.N. Robillard, and M.P. Robillard, "Types of Collaborative Work in Software Engineering", *The Journal of System and Software*, 53, 2000, pp. 219-224.

[15] P. N. Robillard, P. Kruchten and P. D'Astous, *Software Engineering Process with the UPEDU*, Addison Wesley, Pearson Education, 2002.

[16] C. Seaman, *Organizational Issues in Software Development: An Empirical Study of Communication*, Ph.D. Thesis, Computer Science Department, University of Maryland, Technical Report CS-TR-3726, UMIACS Technical Report UMIACS-TR-96-94, 1996.

[17] C. Seaman. "Qualitative methods in empirical studies of software engineering", *IEEE Transactions on Software Engineering*, 25:4, 1999, pp. 557–572.

[18] W. Tichy, "Should computer scientists experiment more?", *Computer*, 31:5, 1998, pp. 32–40.

[19] R.J. Walker, L.C. Briand, D. Notkin, C.B. Seaman, W.F. Tichy, Panel: "Empirical Validation-What, Why, When, and How", *Proceedings, International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 721-722.

An empirical study on Global Software Development: Offshore Insourcing of IT Projects

Rafael Prikladnicki, Jorge L. N. Audy, Roberto Evaristo
School of Computer Science, PUCRS, Porto Alegre, Brazil; University of Illinois, Chicago, USA
rafael@inf.pucrs.br, audy@pucrs.br, evaristo@uic.edu

Abstract

The objective of this paper is to present lessons learned from a case study conducted in a Brazilian software development unit owned by a multinational organization. The focus of this study is to understand the factors that enable multinationals and virtual corporations to operate successfully across geographic and cultural boundaries. Since the number of organizations distributing their software development processes worldwide keeps increasing, this change is having a profound impact not only on marketing and distribution but also on the way products are conceived, designed, constructed, tested, and delivered to customers. Our results show empirical results towards the identification of problems the organizations involved in offshore insourcing of IT projects have faced when going global.

1. Introduction

Software has become a vital component of almost every business. Success increasingly depends on using software as a competitive advantage [1]. More than a decade ago, many organizations began to experiment with remotely located software development facilities (also called Distributed Software Development - DSD) seeking lower costs and access to skilled resources. Economic forces are relentlessly turning national markets into global markets and spawning new forms of competition and cooperation that reach across national boundaries [2].

This change is having a profound impact not only on marketing and distribution but also on the way products are conceived, designed, constructed, tested, and delivered to customers. For these reasons, DSD has attracted a large research effort in software engineering (i.e., [1]; [2]; [3]; [4]; [5]; [6]). The search for such competitive advantage forces organizations to search for external solutions in other countries, and foster the Global Software Development (offshore sourcing). This epitomizes the traditional problems and the existing challenges.

The two main options currently under use include offshore outsourcing (contracting services with an external organization located in another country) as well as offshore insourcing (contracting with a wholly owned subsidiary also located in another country). The first has become fairly common, but difficulties abound in trying to develop a relationship with an unknown foreign partner that is time and geographically distant. Such issues have led select organizations to create their own software development centers in countries like India, Russia, Brazil, Ireland, etc. Although offshore insourcing bypasses some of the tough contracting difficulties found in organizations that are involved with traditional offshore outsourcing, a whole different set of issues is created. And this is what we are trying to address in this paper.

This paper focus on problems that organizations (specifically those involved in offshore insourcing) have faced when going global in software development and how these problems have been addressed. The research question can be defined as: What are the main issues related to the performance of IT projects when developed in an offshore insourcing environment, and how each issue can be addressed?

In order to answer the research question, a case study was conducted identifying some of the difficulties, and solutions involved. This research has as purpose to explore the main issues found in the case study, looking for improvements in projects being developed in this environment. The results are analyzed and the existing challenges identified. Our contributions are the lessons learned from the case study.

This paper has the following structure: section 2 presents the theoretical base; section 3 describes the research method; section 4 describes the case study; section 5 discuss the results found in the case study and presents the lessons learned; section 6 presents the conclusions, future studies and the research limitations.

2. Theoretical Base

2.1. Global Software Development (GSD)

As said by Pressman [7], software process is defined by a set of activities, methods, practices and technologies that people and companies use to develop and to keep related software and products. The interest in the software process is based on the following premises:

- The software quality is strongly dependent on the quality of the process used in its preparation;
- The software process can be defined, managed, measured and improved.

However, even using a well-defined development process it is not a simple task to develop software. As part of the globalization efforts currently pervading society, software project teams have also become geographically distributed on a worldwide scale. This characterizes Global Software Development (GSD).

Organizations search for competitive advantages in terms of cost, quality and flexibility in the area of software development [8], looking for productivity increases as well as risk dilution [9]. Many times the search for these competitive advantages forces organizations to search for external solutions in other countries (offshore sourcing). This epitomizes the traditional problems and the existing challenges in GSD.

GSD causes a profound impact on the way the products are conceived, designed, constructed, tested, and delivered to customers [1]. Thus, the structure needed to support this kind of development is different from the one used in collocated environments. Different characteristics and technologies are needed, increasing the necessity of considering some details not perceived before. GSD has diverse effects on many levels, including strategic issues, cultural issues, knowledge management and technical issues.

Tools and technological environments have been developed over the last few years to help in the control and coordination of the development teams working in distributed environments. Many of these tools are focused in supporting procedures of formal communication such as automated document elaboration, processes and other non-interactive communication channels.

Nowadays, some studies can be found in the literature, proposing some models for global software development. These studies consider both technical and non-technical factors.

2.2. Offshore Sourcing

Offshore sourcing of IT work is increasingly occupying the attention of IT managers in U.S.-based firms. The term "offshore sourcing" includes both offshore outsourcing to a third-party provider as well as offshore insourcing to an internal group within a global corporation [10]. Organizations that avail themselves of outsourcing services can concentrate in its core businesses, potentially reducing the software development team. The combination of these factors results in a significant reduction in time and cost of software development. Insourcing organizations have as advantages the domestic accountability, since they utilize their own resources from the organization software development centers.

IT managers are being pressured to contain costs in addition to ramping up projects quickly, finding experienced staff in fast-moving technologies, and innovating constantly with IT. To acquire the IT competencies that address these challenges, IT managers can choose one of two strategies: either outsource to a domestic supplier or go offshore. The foreign sourcing of IT work is growing based on some reasons:

- Technologies for managing and coordinating work across geographic distances have matured considerably;
- Offshore organizations (both internal and third-party) have improved their software development and project management capabilities.

In the study conducted by Carmel and Agarwal [10], it was identified that the offshore IT sourcing is maturation process and have some stages. The authors proposed four stages in a model called SITO (Sourcing of IT Work Offshore). Each stage in this model is characterized by a set of strategic imperatives and internal firm dynamics.

In the study, the authors understand that technology companies that are in the stage four have different organizational structures and mechanisms than the other companies. The first idea was that these firms have accumulated considerably more experience in offshore IT sourcing, but they usually preferred to own their IT units, and this can lead to many difficulties in terms of software development.

3. Research Method

This research is exploratory in nature based on case study [11]. The case study was developed in a software development unit owned by a multinational organization with worldwide units. The organization works with computer manufacturing and support and is recognized as SW-CMM[1] level 2. It has software development units responsible for **internal client demand** worldwide. Its headquarters are located in the U.S.

The data collection was constituted of primary sources (interviews) and secondary sources (document reviews

[1] SW-CMM is one of the CMM models used for software engineering organizations (http://www.sei.cmu.edu).

and software development process). Considering the respondents, we interviewed 11 people – located in Brazil – from two projects. They represented project team members, development managers, quality assurance team members, software process improvement responsible and individuals representing the organization strategic level. We developed two questionnaires, each considering a specific dimension to be explored: organizational dimension, containing information about the organization as a whole and the strategies involving GSD, and the project dimension, containing information related to the projects selected to be part of this study.

4. Case Study

The case study was developed in the software development unit located in Porto Alegre, south of Brazil. This center aims to perform worldwide technological development for the organization. Almost all projects are distributed, mainly global, since customers and users are located in offices around the world. It has 120 collaborators working in software development and all clients are internal to the organization. The software development process is based on the MSF (Microsoft Solutions Framework), and also on known methodologies, like RUP (Rational Unified Process), PMI (Project Management Institute). The unit studied is recognized as SW-CMM level 2 since January of 2003.

Considering the reasons to invest in global software development, the individual representing the strategic level of the organization pointed out the following items:

- Cost reduction;
- Expanding strategy to global markets;
- Consolidate the organization trademark outside the U.S.;
- Global standard of software development.

The interviews were conducted considering two global projects each one from one department in the organization. For both projects we considered the interaction among (inter-group) project team, users and customers and the interaction inside (intra-group) each group.

5. Case Study Results

5.1. Difficulties found

According to the interviews conducted in the organization, the GSD difficulties are related to the requirements engineering, lack of standards of the activities between distributed teams, the difficulty of share information and the lack of a well-defined software development process. Besides that, corroborated by Carmel [2], and Evaristo [6], there were difficulties concerning language barriers and communication, cultural differences, context sharing and trust acquisition between teams (Table 1).

Table 1. GSD difficulties found

GSD difficulties
Requirements engineering
Software development process
Standards
Communication and language
Culture and context sharing
Trust

Requirements engineering was considered as a continuous difficulty, involving requirements elicitation, analysis, specification, validation, and management. Some individuals interviewed mentioned that since the project is distributed in multiple sites, there is a necessity of having as detailed a set of requirements as possible.

The software development process itself was considered a large difficulty since sometime distributed teams are not using the same process. In addition, software configuration is a critical issue, being the source of many problems related to the development (artefacts with different versions and content in each site).

Communication and language problems were motivated by the cultural differences between both the dispersed individuals and the sites. Finally, trust was also a problem, mainly the necessity of a distributed trust acquisition.

5.2. Solutions

Although there are many possible solutions for each difficulty identified, the organizations focused their solutions mainly on the need for work standardization, investment in planning, and process engagement. It was also mentioned the integration and ways to increase trust between global teams, and continuous training, also mentioned by [6]. (Table 2):

Table 2. Solutions implemented

Solutions
Planning
Training
Standardization
Requirements Engineering
Trust and integration

The initial planning was a necessity identified to select the projects to be distributed, evaluating its characteristics and the unit availability to receive it. Moreover, it was perceived that the process engagement plays an important role to start the interaction between distributed teams.

Another solution implemented was training in soft skills (non-technical factors). Topics explored included leadership, communication, culture, context sharing, project management, and technical training. Standardization was adopted when the distributed teams were not using the same process. Three strategies were considered: forcing standardization; blending methodological components from the various sites into one “new” methodology; and imposing high-level guidelines.

The organization is investing in face-to-face requirements elicitation. But this depends on the project characteristics and travel limitations. There was a big effort in having formal approvals for artefacts in every project. Finally, integration activities are being conducted, aiming at trust acquisition. Some of these activities are developed virtually, but most of them occur when teams (or part of it) meet each other face-to-face.

5.3. Critical Success Factors

The critical success factors identified are directly related to the organizational “modus operandi”. For the same activity we can have different factors, each one related to the strategy adopted by each organization. Consolidating the results of this study, we identified the following critical success factors (Table 3):

Table 3. Critical Success Factors

Critical Success Factors
Software Development Process
Training
Planning and Engagement
Infra-Structure
Team integration
Communication and Feedback

The software development process was considered one of the most important success factors for distributed projects. A large investment in training resulted in an improving relationship. The initial planning was important to evaluate distributed projects correctly and to select the proper unit to receive each project. The process engagement was considered a success factor because it was the first contact between the teams in some projects. Likewise, integration activities were also a success factor because it improved individuals’ soft skills, increasing trust and minimizing cultural differences. Finally, integration improved the communication and feedback.

6. Lessons Learned

The study conducted in the organization shows many characteristics of GSD (section 4). These characteristics were identified based on the difficulties, solutions and critical success factor found and listed previously.

In spite of being an offshore insourcing software development center that has been recently set up (two years), when we examine how the head office and its Brazilian branch conceived and conducted the offshore project development, we can infer a series of lessons.

In this section we will present some of the lessons learned based on the empirical results found.

Table 4. Lessons Learned

No.	Lesson
#1	The existence of a global and well-defined software development process is very important in distributed projects
#2	Requirements engineering is the main challenge for the software development process point of view
#3	The planning phase is important to organize and manage the distributed projects properly
#4	The project management, and in particular risk management need additional effort and steps
#5	The investment in recruiting and training global teams can minimize the difficulties related to the non-technical dimension

Lesson 1: The existence of a global and well-defined software development process is very important in distributed projects

According to Pressman [7], a well-defined process is a process that has a good documentation, detailing what is being done (product), when (steps), for whom (actors), the artefacts used (input) and the developed artefacts (output/results). Moreover, a life cycle must be selected as the starting point for any project.

The study showed that all projects without a well-defined process had many difficulties, some of them related to the process (requirements, configuration management, testing, etc.), and others inherited, as communication, synchronization and trust. Thus, a single and well-defined process in accordance with the project environment can be the solution for many difficulties in global development.

Lesson 2: Requirements engineering is the main challenge for the software development process point of view

Requirements engineering plays an important role in the software development. A requirement is the condition or capacity that a system that is being developed must satisfy [12]. Therefore, the compliance with requirements determines the project success or failure. Requirements are identified, registered, organized and verified during the project development, and are essential to keep the agreements among project team, users and customers.

The problems related with requirements engineering are one of the main reasons for software projects failures [12]. Research has identified [12] that 70% of the requirements were difficult to identify and 54% were not clear and well organized. Therefore, it is not difficult to find errors in requirement specifications with a resulting large impact in the project costs. It is clear that the earlier a problem is detected and solved (especially during the requirements phase) the earlier other problems are minimized in the following phases [12].

Almost all project managers and technical leaders interviewed pointed out difficulties related to requirements engineering activities. One project had the requirements instability as the main problem, mainly because the distance between teams, compromising understanding and agreement between parties. In all projects the requirements were identified as a challenge, involving activities like meetings, requirements documentation as soon as defined, traceability, requirements control and management.

Lesson 3: The planning phase is important to organize and manage the distributed projects properly

To define strategies of an organization in the information systems area based on a formal planning process is a challenge [13]. The lack of a formal planning phase can be one of the main problems before the software development process. According to [14], the lack of a formal planning phase causes a great number of problems in the next phases.

In the study, it was identified the initial planning as a formal and basic phase to decide if a project has characteristics to be distributed and to plan its development. Thus, the planning basically involves the definition of the strategies, which will lead the development of the whole process. Based on the case study, it is possible to consider the planning phase as a former cycle of many projects cycles derived from the planning process.

Lesson 4: The project management, and in particular risk management need additional effort and steps

According to the Project Management Body of Knowledge [15], project management is the application of knowledge, abilities and techniques to plan activities that can reach the needs and expectations of all stakeholders involved in a project. Bad project management can mean the loss of the project and the resources involved. Therefore, risk management is one of the most important activities in a project, involving the identification; treatment and elimination of risk sources before it become a concrete threat for the project. Risks can also be treated in different levels.

In the study, all activities involving project management and risk management have a huge importance for distributed projects and the managers interviewed said that in distributed projects these activities take longer than in traditional projects (collocated), requiring a larger effort and some additional steps in the traditional models. Additionally, all risks concerning the decision of sending a project to be developed offshore were considered in the project risk management.

Lesson 5: The investment in recruiting and training global teams can minimize the difficulties related to the non-technical dimension

In global development, project managers have to organize and manage projects with a team composed by individuals from different cultures, with different customs. According to Kiel [16], the technical barriers are diminishing rapidly. On the other hand, the human factors are less studied. Therefore, when distribution ultimately fails, it can be a web of social, cultural, linguistic and political factors, rather than use or misuse of specific tools or techniques [16]. There are other factors that can be added to this list (communication, context, interpersonal relationship), but this study brought a very important conclusion. The lack of investment in the recruiting and training of project teams to be global teams can lead to unexpected problems in the project development.

Organization’s policy included investing in team training, focusing communication, cultural differences, trust, and context sharing. As a result of this initiative, the interactions between distributed teams (including customers, users and project team), were easier. Problems identified before the training started to occur less frequently, showing that the management of distributed teams is a key to the project success.

7. Conclusions

Any software professional knows that even collocated software development is fraught with difficulties. The entire field of software development, or software engineering, is still maturing. It is becoming harder to justify completing a software development project inside company walls.

As the software community appreciates the economy of merging diverse development skills and domain expertise, and as communication media become more sophisticated, the cost and technology pushes more companies toward global software development. It is becoming less and less cost-effective or competitive to develop a software product in the same building, company, or even country.

Improvements in tools and methods over the last several decades are allowing groups from different locations and backgrounds to come together as a global software development team. Moreover, GSD is leading the researchers to acquire new knowledge and to be more interdisciplinary.

This paper advances the knowledge in the GSD area by identifying important characteristics of this recent and growing field, focusing on the offshore insourcing of IT projects. We discussed lessons learned based on a case study in a software development unit from a multinational organization. These sets of results give us indication that the search for greater formalism and the selective utilization of international patterns will provide full conditions to overcome the problems originating from the dispersion specifically in the case of wholly-owned subsidiaries. Planned follow up studies in this topic will continue to analyze the organizations difficulties and solutions and will going deep in the study of specific factors found in this work, like requirements engineering, risk management, and project allocation, despite of analyzing how other organizations in similar situations are dealing with all these problems.

Finally, this project is not only a landmark study in the area of offshore insourcing, something until recently not been researched, but also has strong implications to the more traditional offshore outsourcing. The key reason is that most of the work currently being done in offshore outsourcing is seen under the perspective of contracting; although obviously very relevant, eventually such studies will need to go further past that issue – which is exactly what we are proposing to do in the near future.

8. References

[1] Herbsleb, J. D., and Moitra, D. “Global Software Development”, IEEE Software, March/April, USA, 2001, p. 16-20.

[2] Carmel, E. “Global Software Teams – Collaborating Across Borders and Time-Zones”. Prentice Hall, USA, 1999, 269p.

[3] Karolak, D. W. “Global Software Development – Managing Virtual Teams and Environments”. Los Alamitos, IEEE Computer Society, USA, 1998, 159p.

[4] Damian, D. “The study of requirements engineering in global software development: as challenging as important”, Proceedings of International Workshop on Global Software Development at ICSE, Florida, USA, 2002.

[5] Prikladnicki, R.; Audy, J. L. N.; Evaristo, R. “Distributed Software Development: Toward an understanding of the relationship between project team, users and customers”. *Proceedings of ICEIS*, Angers, France, 2003.

[6] Evaristo, J. R., Scudder, R., Desouza, K. and Sato, O. "A Dimensional Analysis of Geographically Distributed Project Teams: A Case Study," forthcoming in the Journal of Engineering Technology and Management, 2003.

[7] Pressman, R. S. “Software Engineering: A Practitioner’s Approach”. Fifth Edit, USA, 2001.

[8] Prikladnicki, R., Peres, F., Audy, J., Móra, M. C., and Perdigoto, A. “Requirements specification model in a software development process inside a physically distributed environment”, Proceedings of ICEIS, Ciudad Real, Spain, 2002.

[9] McConnel, S. “Rapid Development”. Microsoft Press, Canada, 1996.

[10] Carmel, E.; Agarwal, R. “The Maturation of Offshore Sourcing of Information Technology Work”, *MIS Quarterly Executive*, Vol. 1, No. 2, June 2002, 65-77.

[11] Yin, R. K “Case study research: design and methods”, Sage, USA, 1994.

[12] Oberg, R., Probasco, L., and Ericsson, M. “Applying Requirements Management with Use Cases”, Rational Software White Paper, Cupertino, CA, USA, 2000.

[13] Audy, J. L. N. “Strategic Planning Model of Information Systems: contributions of decision process and organizational learning (in Portuguese)”. Ph. D. Tesis, PPGA – UFRGS, Brazil, 2001.

[14] Martin, J. “Information Engineering (in Portuguese)”. Rio de Janeiro, Campus, 1991.

[15] A guide to the project management body of knowledge (PMBOK guide). Project Management Institute, USA, 2000. 216p.

[16] Kiel, L. “Experiences in Distributed Development: A Case Study”, Proceedings of International Workshop on Global Software Development at ICSE, Oregon, USA, 2003, 4p.

Analyzing Intercultural Factors Affecting Global Software Development – A Position Paper

Philippe Kruchten
University of British Columbia
Vancouver, Canada
pbk@ece.ubc.ca

Abstract

This position paper presents the efforts we have undertaken to study the impact of intercultural factors on global software development projects. A bottom-up approach looks at the effect of individual intercultural factors on software practices, while a top-down approach strives to identify positive or negative organizational and behavioral patterns.

1. Introduction

Global software development projects may succeed or fail for reasons that have nothing to do with the technology, with the time differences, the (tele-) communications mechanisms used, or the product being built, but because of subtle intercultural factors. The issues at stake are not superficial matters of ways of dressing, working, speaking, in small daily behaviors, but are founded in the fundamental differences in the systems of values that govern our lives. A first step that global organizations have taken in the last 15 years was to raise the level of awareness of their employees world-wide on the cultural differences, through various programs of intercultural or diversity training. But cultural awareness is not sufficient to overcome many of the obstacles that cultural differences bring in the way of global project success. We have started two efforts: first, to take a more systematic look on how intercultural factors affect positively or negatively the outcomes of software development practices. Second, to identify patterns and anti-patterns (i.e., patterns with negative effects) of organizational behavior that impact the outcome of outsourcing or off-shoring of software development projects.

2. Global software development

A lot of attention has been drawn on the outsourcing or off-shoring phenomenon, in particular with the successes of Indian software companies. An estimated half a million jobs would have “fled” from North America to India by 2015 [17]. This is not just pure tabloid hype: I have friends in Vancouver who have lost their software development jobs to … some other friends in Bangalore. IT projects are the second largest class of outsourced activities after call centers.

Most of this attention has been on the economic aspects, on the labor issues, and as well as on the communication mechanisms and tools [4], less on the processes [6], little on culture [16, 18]. A great deal has been published on how to behave or not to behave when doing business in this or that country. While useful and accurate, they often completely lack any depth and analysis of the fundamental mechanisms at play.

Only recently have a few researchers started to look at the specific issues of intercultural factors on technical professions and global projects: Laroche [14], Carmel [4], Karolak [13], Schneider and Barsoux [20]. Most of the work published today keeps referring to Hofstede [10], a study on a large population, indeed, but now almost 40 years old, and performed inside one single company, IBM.

3. Overall approach

The first part of the study is to identify the *impacts of intercultural factors on software development practices*. The overall approach for this study is as follows:

1. Identify and sort out intercultural factors
2. Identify and sort out a set of practices, representative of software engineering
3. Identify interesting cultural groups and their profile on the selected set of intercultural factors
4. Using expert advice, literature studies, and possibly surveys, make a first attempt at identifying pairs [practice + intercultural factor] that are significantly affected.
5. Then, for some elements of this “hot” list of affected practices, set up experiments to validate and quantify the effect.
6. Or use post-mortem analysis of real-life projects to identify occurrences of affected practices

In parallel, proceed with some case studies of outsourced or global projects, looking at outcomes, lessons learned and doing a root cause analysis. It could also provide the basis for point 6 above

The second part of the study is to identify *behavioral patterns* that enhance or hinder the outcome of global projects.

The method used will combine ethnographic studies, with content analyses, surveys, experiments, trying to avoid ethnocentrism in the study itself [23], and not to lose of the specific "emics" elements of a culture.

4. Intercultural factors, or variables

As the primary source of intercultural factors or variables, we are using the classic works of Edward T. Hall [9], Geert Hofstede [10, 11], Alan Fiske [6, 7], and Fons Trompenaars & Charles Hampden-Turner [25].

4.1 Edward Hall: Beyond Culture

One of the pioneers of this field, Edward T. Hall has looked at communications, and discriminates cultures on high context and low context communication. Hall also looked at the way cultures handle time—monochronic cultures (M-time) versus polychronic cultures (P-time). Hall also has plenty of other interesting observations on situational dialects, actions frames, and education.

- Low-high context
- M-time and P-time

Other ideas of Hall about physical distance between individuals, what he calls *proxemics*, may not be too useful in the context of global development.

4.2 Hofstede: Groupthink

Although Hall's work is based on his own observations—he had lived with several tribes in the US Southwest (Hopi, Navajo), and in several countries in Asia—the Dutchman Geert Hofstede took a completely different approach. He was given access to a vast amount of data, uniformly collected across tens of thousands of employees of a large multinational company (IBM) in the late 1960s and 1970s, and he used sophisticated (at the time) multivariate analysis to extract and then interpret major discriminating factors across cultures, crudely defined by country.

Here are the five views he came up with, and compared two by two:

- Power distance
- Collectivism versus individualism (see also [24])
- Femininity versus masculinity
- Uncertainty avoidance
- Long-term versus short term orientation

4.3 Trompenaars & Hampden-Turner: Reconciling the opposites

Similarly to Hofstede, these two researchers have defined a slightly different set of discriminating factors, based on the studies they've done as part of a consulting practice for large multinational companies. They too distinguish several "views":

- Universalism vs. particularism
- Individualism vs. communitarianism
- Neutral vs. emotional
- Specific vs. diffuse
- Achievement vs. ascription (attitude toward titles, degrees,…)

And a few secondary ones, such as:

- Attitude to time
- Attitude to the environment (i.e., nature)
- Gender, race, class, religion

Less known than Hofstede's, their factors may prove more usable to analyze a business situation.

4.4 Fiske: Four elementary forms of sociality

- CS: communal sharing: do people treat all members of a category as equivalent.
- AR: authority ranking: do people attend to their positions in a linear ordering.
- EM - equality matching: how people keep track of the imbalances among them.
- MP: market pricing, how people orient to ratio values.

This is a large number of factors. To reduce the spectrum of possibilities offered by a wide range of intercultural factors, we may be able to exploit the concept of synthetic culture profiles introduced by Gert Jan Hostede (Geert Hofstede's own son) in [12]. This would also avoid polarizing on anecdotes and stereotypes ("Japanese vs. American", "Brits vs. Greeks").

5. Software practices

The software engineering practices that are likely to be affected are not so much the ones fully supported by machines, automated, or the repetitive, human-intensive ones, or the ones close to the code or to the bits. The practices affected are the ones that involve human to human communication, either at the time they are performed, or later, in their consequences. Some would say: "this is covering most of what we do in software". Not quite. If we looks at the systematic "CMM level 3" type of software processes used in global outsourcing projects, a lot of the nitty-gritty daily work is specified there, and does not involves too much human interaction. We can certainly look at how these processes are themselves tainted by the cultural backgrounds of their authors (and I am looking at the Rational Unified Process [15] with that critical eye).

5.1 Agile practices

To find more likely candidates we may look at the agile set of methods and practices [2], which precisely have come to rely much more on direct person-to-person interaction and less on "follow the plan", "fill the template", and "check the boxes" approaches.

The twelve XP practices [3] constitute a good representative set:

- Collective ownership
- Planning game
- Pair programming
- Customer interaction
- Whole team. *Etc.*

We should add the practice of:

- Scrum [21].

Unfortunately these practices are often confined within a single, co-located (and therefore often culturally homogeneous) team and they are not visible at the hinges between two cultures in global projects. One exception however is the interaction with the customer (see §6.2)

5.2 Other practices

- Reviews, inspections and walkthrough
- Retrospectives and post-mortem, process improvement process
- Wideband Delphi, and other approaches using expert knowledge
- Planning and estimation, especially scheduling
- Management milestone and other "critical" decision-making meetings (Project Review Authority, Change Control Board, etc.)
- Performance reviews, and other HR processes
- Organizational structure, and communication

The matrix of [factors x practice] is quite large. Some clustering maybe necessary, identifying groups of practices that are affected in similar ways, and maybe using one of them at the canonical representative.

6. Examples

To illustrate the approach, here are two [factor, affected practice] pairs and one pattern.

6.1 Reviews and chronicity

Several impacts have been identified, for example by Laroche [14]. One such impact he calls:

"*time is up*: M-time people tend to end the meeting or conversation at the scheduled end-time, P-time people tend to end when the conversation runs out of steam and rarely at the scheduled end time. When they work together, polychromic people may think that the meeting ends abruptly, before they have a chance to say their whole piece. In contrast, M-time people may consider that polychronic meetings go on past the point of effectiveness."

Laroche identifies several other issues: agenda (implicit or explicit), etc.

Example of occurrence: Quebecers working with Ontarians, or Spaniards with Germans. Note that none of the party would either deny the benefits of a review, or challenge the process, and the mishaps are independent of the actual technical issues raised.

6.2 Requirement management and power distance

Thanasankit and Corbitt have studied the factors of power distance and uncertainty in Thai culture [22]. These factors contribute towards hierarchical forms of communication and decision making processes in Thailand, especially during Requirements Engineering. Their research shows that the decision making process in Thailand tends to take a much longer time, as every stage during Requirements Engineering needs to be reported to management for final decisions. The tall structure of Thai organizations also contributes to a bureaucratic, elongated decision-making process during information systems development. In eliciting/validating/prioritizing requirements, often who said what and where that person seem to appear in the hierarchy is more important than the needs or the technical issues.

6.3 The proxy pattern

More efficient than across the board intercultural training, hoping that all will behave in a harmonized and cultureless fashion, some organizations have found ways to exploit the talent of very rare individuals, which are used as proxies. Their life story has made them "bi-coded" as a colleague calls them: able to operate equally at ease in two different cultures.

For example, a typical proxy was born and raised in Asia, came to North America to study, stayed some 6 to 8 years, returned to his country, had a quick and rather successful career, and then returned to North America to man a "beachhead" of outsourcing. The proxy operates relative to his company as a true full-fledge citizen, but he also has internalized the values and associated behaviors of North American high tech culture, and actually spends most of his or her time doing some "impedance adaptation" between the two cultures.

There is a related “anti-pattern.” Not everybody can play the role of the proxy. If an individual has not assimilated completely the 2 cultures, and is for example promoted from Asia to a position of proxy in North America merely as a perk, as an award for good performance at home, that person may effect more damage in the relationships between supplier and purchaser of outsourcing.

7. Conclusion and Future Work

There is not much to conclude, this early in our study. My hope is that a systematic look at impacts and at patterns will give us insights on how to describe, express, configure and enact software engineering processes for global software development, in ways that respect the specific cultures of all nations and groups involved, or that even take advantage of the strength of certain groups.

Thanks to Mackie Chase and Leah McFaddyen of UBC’s Center for Intercultural Communications for pointing me to useful things to read.

References

[1] N. J. Adler, *International Dimensions of Organizational Behavior,* 3rd ed. Cincinatti, OH: South Western College Publ., 1997.

[2] Agile Alliance, *Manifesto for Agile Software Development*, 2001. http://agilemanifesto.org/

[3] K. Beck, *Extreme Programming Explained: Embrace Change.* Boston: Addison-Wesley, 2000.

[4] E. Carmel, Global Software Teams: Collaborating Across Borders and Time Zones. Upper Saddle River, NJ: Prentice Hall, 1999.

[5] E. Carmel, "Tactical Approaches for Alleviating Distance in Global Software Development," *IEEE Software*, pp. 22-29, 2001.

[6] A. P. Fiske, *Structures of Social Life: The Four Elementary Forms of Human Relations.* New York: Free Press (Macmillan), 1991.

[7] A. P. Fiske, "The Four Elementary Forms of Sociality: Framework for a Unified Theory of Social Relations," *Psychological Review*, vol. 99, pp. 689-723, 1992.

[8] A. Gopal, T. Mukhopadhyay, and M. S. Krishnan, "The role of software processes and communication in offshore software development," *Commun. ACM*, vol. 45, pp. 193-200, 2002.

[9] E. T. Hall, *Beyond culture*. New York: Anchor Books/Doubleday, 1976.

[10] G. Hofstede, *Culture's consequences.* Beverly Hills, CA: Sage, 1980.

[11] G. Hofstede, *Culture and organizations--Software of the mind.* McGraw-Hill, 1997.

[12] G. J. Hofstede, P. B. Pedersen, and G. Hofstede, *Exploring culture; exercises, stories and synthetic cultures.* Yarmouth, Maine, USA: Intercultural Press, 2002.

[13] D. Karolak, *Global Software Development : Managing Virtual Teams and Environments.* NY: Wiley, 1998.

[14] L. Laroche, *Managing Cultural Diversity in Technical Professions*. Butterworth-Heinemann, 2002.

[15] P. B. Kruchten, *The Rational Unified Process: An Introduction*, 3 ed. Boston: Addison-Wesley, 2004.

[16] J. S. Olson and G. M. Olson, "Culture Surprises in Remote Software Development Teams," *ACM Queue*, vol. 1, pp. 52-59, 2004.

[17] D. H. Pink, "The new face of the silicon age," in *Wired Magazine*, Feb. 2004, pp. 94-103, 138.

[18] M.M. Sathyarnarayan, *Offshore Development—Proven Strategies and Tactics for Success*, Cupertino, CA: GlobalDev Publ., 2003

[19] E. H. Schein, *Organizational culture and leadership,* 2 ed. San Francisco: Jossey-Bass, 1992.

[20] .S. C. Schneider and J.-L. Barsoux, *Managing Across Cultures*. Prentice-Hall, 2003.

[21] K. Schwaber and M. Beedle, *Agile Software Development with SCRUM.* Upper Saddle River, NJ: Prentice-Hall, 2002.

[22] T. Thanasankit and B. J. Corbitt, "Thai Culture and Communication of Decision Making Processes in Requirements Engineering," *Proceeding of CATAC’00 Cultural Attitudes Towards Technology and Communication*, Perth, 2000.

[23] H. C. Triandis, *Culture and Social Behavior*. New York: McGraw-Hill, 1994.

[24] H. C. Triandis, *Individualism and collectivism.* Boulder, CO: Westview Press, 1995.

[25] F. Trompenaars and C. Hampden-Turner, *Riding The Waves of Culture*, 2 ed. New York: McGraw-Hill Trade, 1997.

The Benefits and Limitations of Knowledge Management in Global Software Development

Torgeir Dingsøyr[1], Knut-Helge Rolland[2], M. Letizia Jaccheri[2],
[1]*SINTEF ICT,* [2]*Dept. of Computer and Information Science NTNU,*

torgeir.dingsoyr@sintef.no, *{knutrr|letizia}@idi.ntnu.no*

Abstract

The role of knowledge management practices and tools in global software development will be explored by empirical investigations. These investigations will look at global software development processes by taking into special account multicultural factors and will rely on both quantitative methods for project selection and qualitative methods for in depth study of the single project contexts.

1. Introduction

Software development work is becoming global in the sense that development work is increasingly carried out in teams that are geographically separated across national boundaries and cultures. This trend is captured in commercial software production where parts of the development activities often are outsourced to low-cost countries, as well as in 'open source software' projects where development sometimes is global in scope.

The problem we want to discuss in this position paper are the fact that while software development becomes increasingly distributed and global in nature, much of the techniques and tools for improvement still assume that individuals are co-located. For example, the principle of pair-programming in the XP approach was established to improve learning and knowledge-exchange among programmers [2] and traditional code inspection methods often assume face-to-face interaction in terms of more or less formal meetings.

Software development conducted in a distributed fashion is often referred to as 'Global Software Work' (GSW). More accurately, Sahay defines GSW as "software work undertaken at geographically separated locations across national boundaries in a coordinated fashion involving real time and asynchronous interaction" [9]. Given these characteristics, it becomes clear that GSW involves different kinds of complexities compared to traditional software development where members of the project are more or less co-located and are therefore able to share their experiences through face-to-face communication.

Drawing from past experience reported in the literature on global software development, we can single out some factors that are of profound relevance. Previous studies have shown how cultural issues are directly related to and influence how software is developed and managed. It has been shown that teams from different cultures tend to prefer dissimilar approaches to architectural design. For example, there are differences in how abstractions are chosen and what architectural patterns are used [3]. Clearly, an understanding of these issues would be of profound importance for managing global software projects successfully. Moreover, recent research has underscored the challenge of adopting common and standardized practices and tools in global software work and the need for developing a specific competence to do global product development [8].

However, while the above literature points at some practices and competencies for successfully conducting global software development, little is said about how such competencies can be learned and adopted in other organizations and domains. Considering the limited capability of learning in many software development projects and organizations [6], it is then important to investigate how organizations and individuals can facilitate learning both within specific global software development initiatives and between different projects.

In this position paper we will outline a research design for studying the limits and benefits of Knowledge Management (KM) practices and tools for improving organizations' capabilities of continuous learning in global software development projects.

2. Arenas to learn from

Global software development has existed in some forms since the early eighties in both volunteer contexts and commercial ones, such as banking applications. One example of an environment that does global software development is the Open Source community [5]. The Open Source community consists of a myriad of projects, which vary in number of developers and their organization roles, kind and size of developed software, degree of involvement of commercial actors, popularity, vitality, degree of success, and duration of the project. The slogan "the success of open sources projects" seems to stem from the big success of projects such as Linux and Apache [7].

The software research community has devoted a lot of attention to the open source world, as can be deduced for example from the series of open source workshops in the ICSE context. Many are the questions which are of interest of the software community when looking at the open source world. Which are the successful projects and how do we define this notion of success in a not profit world? Is success a function of project vitality? Is success a function of the impact the open source software has on the commercial market, such as for example the Linux operating system?

Given that successful open source project are the focus of our interest, how and what do we want to learn from them and how do we want to transfer the learned knowledge into commercial contexts or even into educational ones? One possibility is that of making hypotheses about the reasons or causes for success of open source projects, and then trying to describe these reasons in order to disseminate them. How does Open source project implement knowledge management? Is it the software process model of open source project which is the cause of success? If we regard a process model as descriptions of tasks, practices, responsibilities, tools, and document types, which of these elements is of most importance for project success?

3. Knowledge Management: Benefits and limitations

Learning in the context of software development has often been limited to different kinds of information technology support for learning and knowledge-exchange [4]. For example, there has been much focus on reusing life-cycle experience, processes, and products for software development in terms of having an 'Experience Factory' [1]. Likewise, the information systems literature has emphasized introduction of 'Knowledge Management Systems' in order to support organisation-wide knowledge-exchange and learning. Arguably, these technologies and knowledge-sharing practices can play important roles in facilitating learning in software organizations. However, as there exists a wide range of different KM practices and tools, there is thus a growing need for investigating empirically what kind of KM that is relevant for global software development. A salient point related to global software development is also that systematic practices and tools for KM are perhaps even more relevant for software development in (globally) distributed settings there lack of more informal face-to-face interaction must be substituted with other ways of coordinating work and ways for facilitating mutual learning between distributed development teams. On the other hand, establishing KM practices and tools across different cultural settings might also be especially challenging due to cultural differences in how knowledge is formed, structured and utilized in different countries [10].

Thus, the literature seems to suggest that there are both potential benefits and limits for improving learning through KM in global software development. Research should be conducted in order to illuminate these benefits and limits in more detail, and for increasing the understanding of global software development in general.

4. Research design

In order to study the new landscape of software development, it is relevant to draw from different research disciplines and perspectives in a cross disciplinary and multi-perspective approach. On the research method side, we will be open to both the empirical software engineering community and the community of research that focuses on explaining software development in relation to a broader social, organisational, and economical context.

Our investigations will look at global software development processes by taking into special account multicultural factors. On the one hand, there is a need to use descriptive statistics to get an orienteering map in the complex world of global software development. If we look at open source projects for example, it is of great help to classify them by evaluation parameters, like number of active users, lines of code, age, vitality, number of represented countries which help us to compare them and to choose those which we want to study in depth.

In this way, we will select a couple of case studies to capture and describe learning practises and networks in real-world settings, how tools, techniques, or concepts are employed. One case will involve Open Source development, and one will involve global software development in a commercial setting. We will select cases involving software development environments in at least three countries, and projects that run over a period of at least two years.

Data from the case studies will be collected according to the following two principles:

- Multiple sources of evidence. We will collect data from several sources, such as documentation, archival records, interviews, direct observations, participant-observation, and physical artefacts. Analysis will be assisted by using the qualitative data analysis tool Nvivo.
- Case study database. We will document the data collected in the case studies in terms of notes, documents, tabular materials, narratives, photographs, and video, and organise it in a case study database. For this purpose we will use the facilities offered by eRoom, which provides a shared, secure workplace on the Web for the project team.

5. Conclusion

In this position paper, we have argued that software development is increasingly global, which makes the complexity of software development larger due to changing technologies, methods, geographical location and multicultural arrangements. Knowledge management tools have the objective to reduce the problems of complexity in global software development. Through investigations of practices in organizational learning, we seek to reach a better understanding of how knowledge management tools and learning issues in general function in global software work. Such an understanding can give the software engineering community better abilities to see what kind of knowledge management practices and tools that are suitable for global development, as well as new insights on key practices from the open source community.

8. References

[1] V. R. Basili, Caldiera, G and Rombach, H.D., "The Experience Factory," in *Encyclopedia of Software Engineering*, vol. 1, J. J. Marciniak, Ed.: John Wiley, 1994, pp. 469-476.

[2] K. Beck, *Extreme Programming Explained*: Addison-Wesley, 190 pp., 2000.

[3] G. Borchers, "The Software Engineering Impacts of Cultural Factors on Multi-cultural Software Development Teams," presented at 25th International Conference on Software Engineering (ICSE'03), 2003.

[4] T. Dingsøyr and R. Conradi, "A Survey of Case Studies of the Use of Knowledge Management in Software Engineering," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, num. 4, pp. 391 – 414, 2002.

[5] J. Feller and B. Fitzgerald, *Understanding Open Source Software Development*: Addison Wesley, 211 pp., 2002.

[6] K. Lyytinen and D. Robey, "Learning Failure in information systems development," *Information Systems Journal*, vol. 9, pp. 85-101, 1999.

[7] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, num. 2, pp. 309-346, 2002.

[8] W. J. Orlikowski, "Knowing in Practice: Enacting a Collective Capability in Distributed Organizing," *Organization Science*, vol. 13, num. 3, pp. 249-273, 2002.

[9] S. Sahay, "Global software alliances: the challenge of 'standardization'," *Scandinavian Journal of Information Systems*, vol. 15,, pp. 3-21, 2003.

[10] G. Walsham, "Knowledge Management: The Benefits and Limitations of Computer Systems," *European Management Journal*, vol. 19, num. 6, pp. 599-608, 2001.