A Case Study of Feature Based Awareness in a Commercial Software Team and Implications for the Design of Collaborative Tools

by

Luis Guillermo Izquierdo Rojas
B.Sc. National University of Cajamarca, 1998

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in the Department of Computer Science

A Case Study of Feature Based Awareness in a Commercial Software Team and Implications for the Design of Collaborative Tools

by

Luis Guillermo Izquierdo Rojas
B.Sc. National University of Cajamarca, 1998

Supervisory Committee

Dr. Daniela Damian, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Departmental Member

Dr. Janice Singer, (Department of Computer Science)
Departmental Member

Dr. Gail Murphy, (Department of Computer Science, University of British Columbia)
External Member

Supervisory Committee

Dr. Daniela Damian, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Departmental Member

Dr. Janice Singer, (Department of Computer Science)
Departmental Member

Dr. Gail Murphy, (Department of Computer Science, University of British Columbia)
External Member

Abstract

Software development is a process in continuous evolution. This characteristic implies also continuous changes in the functionality of the system under development. Some of these changes may cause problems when they are not properly and timely propagated to the project members. The aim of our research is to obtain a good understanding of problems caused by the lack of awareness of changes to features during a distributed software development project, to identify information and artifact repositories used by contributors, and then to draw the requirements of an awareness mechanism to tackle the awareness problem. In order to accomplish our research

goals, we conducted a four month long case study at IBM Ottawa Software Lab, in which we observed the collaboration patterns of a multi-site development project team.

Our findings helped us identify the most important communication media that support development. In particular, we observed that the 34% of communication was by phone and via face-to-face interactions, and email was mostly used to alert contributors about changes to features. We also found that changes were not properly and timely propagated due to different corporate cultures of the project teams. Finally, we found that a high volume of communication makes developers prone to overlook important information that can lead to the generation of errors during development. These findings led us to believe that miscommunication and non-timely communication of changes related to feature development caused the release of code that created failures in stable builds.

To address this problem, we developed the concept of a relationship to link developers to features. Using this concept, we have designed a feature-based Awareness Mechanism System to collect information, create relationships and deliver awareness information to the contributors involved in the implementation of a feature.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First, I would like to thank Dr. Daniela Damian for believing in me and giving me the opportunity to be involved in this research; her continuous support and encouragement helped me to succeed in this research endeavor.

Thank you to Janice Singer for her guidance in learning research methods.

Thank you to my committee for their comments, support and interest on my research.

I would also like to thank to all members of the Team Tools Project - Ottawa, in particular to Jean-Michel Lemieux, and Jim de Rivieres for their guidance and time provided; also I would like to thank the following project members: Heather Fraser-Dube, Jonh Camelon, Rafael Chaves, Andreas Meissner, Carol Yutkowitz, and James Synge for their support and contribution, and Marcellus Mindell for making this research possible.

Funding for my work was provided by NSERC scholarship and IBM Centre for Advanced Studies. I must also thank these funding institutions for supporting my

research.

Thank you to my mother, friends and lab colleagues for their continuous support of my work, and my studies.

Finally thank you to Kaily Stevens and Antonieta Falconi for their editorial assistance and friendship.

Dedication

To my mother

Chapter 1

Introduction

The evolution of a project is encompassed by continuous changes; they are present in almost every kind of human endeavor. Irrespective of a project's nature, changes might reshape the original vision and scope of it, especially if changes come from the end-users of products or services that are under implementation. Being alerted to changes that happen during a development process make developers aware of implications in the final deliverable; hence, depending on how critical the changes are, the developers have to assess risks of the new implementation before proceeding. Timing becomes a relevant factor when we talk about changes. For example, a change done in the structural design of a building after the structure has been built will mean a higher cost than originally budgeted, after implementing the changes.

Software is built in dynamic environments characterized by constant changes to

requirements, design specifications, code, test cases, and other artifacts. Changes are performed by contributors[1] of the system; new changes either can improve the functionality of the system or they can cause defects that will impact the system's performance. If changes introduce defects that are not properly tackled there will be considerable cost to fix them. Previous research has shown for example, that refinements of requirements are made by individuals, but that these requirements are too slowly articulated to enable other contributors in the development process to react to them [11].

In the software development process contributors often play diverse roles of designers, coders, testers, etc. They work in workspaces [2] that are often separate from each other and create software artifacts that are stored in separate repositories. Often the relationship between these workspaces and artifacts is not made explicit. For example, requirements or features may be manipulated in a requirements repository such as Rational RequisitePro, design diagrams may be manipulated in MS Visio, code is edited and written using the Eclipse IDE, and commits are stored in a CVS repository. Testers write their test cases using Eclipse and these are stored in a different repository. However, changes in artifacts stored in different repositories may be made using different tools without a mechanism to propagate changed information

[1] A contributor is a generic term for designer, developer, manager, etc.

[2] A workspace is a space or container that contains all the software artifacts needed by a contributor.

to related artifacts and contributors working on these artifacts. An artifact is related to other artifact(s), when they are part of the implementation of the same system feature.

The variety of tools, artifacts, and repositories generates an environment where change management becomes complex. The higher the number of tools, artifacts, and repositories in which these are stored, the lower the probability that changes will be propagated effectively across the developers' workspaces where these changes need to be communicated. For example, currently the Eclipse IDE does not include a mechanism to alert a developer if a change has been made in repositories other than the CVS code repository. Thus, if a change is made in a requirements repository, Eclipse can not access it, identify the change, and relate it to the code that implements it. Furthermore, while time and distance are important factors in geographically multi-distributed development settings, they are also important factors to consider when a change needs to be propagated between workspaces that are remotely located.

Software development requires intense collaboration between members of a team. Collaboration heavily relies on communication processes among stakeholders. Proper and timely communication allows developers to be aware of changes in remote workspaces that could affect their work.

1.1 Problem Statement

Developers lack timely awareness of changes that affect their workspace. One of the consequences of this problem is that developers become aware of changes in software artifacts such as requirements, design documents, code, and test cases, long after they have been made. This results in lost time because the code that is related to the changed artifact undergoes development even though the artifact had already been changed.

Considering the fact that man-hours is one of the main criteria used to calculate the cost of a project, project costs are bound to increase when the developers responsible for implementation waste time working on code that has already been changed. Companies consequently lose money re-implementing functionality.

1.2 Research Goal

Awareness in software development is a complex problem to solve. This research is the beginning of a large project that will tackle the problem of awareness during the software development cycle, in order to improve the collaboration of software teams. Part of the awareness problem includes the lack of a good understanding of how to support awareness of change in a software development process. It needs to be determined which information needs to be propagated, which artifacts are involved

in the process and how contributors need to be alerted to changes that are relevant to their work.

The short term research goal covered in this thesis is to further our understanding of awareness in software development and the investigation of problems caused by the lack of awareness during the software development process. Our aim is to identify collaboration patterns, software artifacts, and data repositories used by contributors in the development and ways in which they relate to concepts of awareness.

A good understanding of the awareness problem is necessary to the design awareness mechanisms to support the propagation of changes to features to contributors associated with the implementation of a particular feature. The awareness mechanism will alert contributors to changes in the features which will affect their workspaces.

In SEGAL[3], our long term research goal is to implement a feature-based awareness mechanism system to support the collaborative activities of contributors. We will design a mechanism that provides awareness of upstream artifacts, relevant to the developer's workspace and helps developers to stay up-to-date with changes to software artifacts that are related to their work. We believe that providing timely access to the rationale, which is a requirement specification, and current changes related to developer's work (either on demand or through a notification system) leads to improved collaboration in software development. This ultimately leads to better

[3]Software Engineering Global interAction Lab, website: http://segal.cs.uvic.ca

management of changes, improved productivity and higher software quality.

1.3 Approach

We have limited information about how the lack of awareness causes problems in real projects, and our understanding of how to design an awareness mechanism system (AMS) is limited as well. Therefore, we need to understand and characterize the awareness problem in real projects, learn more about the role of awareness in industrial settings and identify the requirements for an AMS. In order to accomplish this, our research approach is as follows:

We performed a review of background and related literature to obtain a good understanding of the awareness concepts and to find what other researchers have done in providing solutions to tackle awareness problems. Most of the information found was related to previous academic research, and case studies done in universities and open source projects.

We also found that commercial tools such as Jazz [12], Composent [46], and Stellation[4] provide only awareness of changes at the code level where code is linked to the owner. However, these tools do not consider the creation of links with other artifacts such as requirements, design documentation and bugs. In this phase of our research we were able to learn theoretical principles to design an awareness system.

[4]Stellation Home Page, http://domino.research.ibm.com/synedra/synedra.nsf

We carried out a case study in which we observed an industrial development practice. We wanted to identify problems generated by the lack of awareness and draw the requirements of an Awareness Mechanism System. At this stage we believed that an automated mechanism could provide better awareness of changes. However, we did not know the requirements of the system. During our research, we systematically examined collaboration patterns involved in the development of a certain feature, project artifacts, and information repositories at an IBM development centre in Canada. We observed a group of developers, collected and processed information. We defined "the number of broken builds produced by the lack of awareness" as our success criterion. We focused our case study on the observation of one major core component (CSM). We selected it because it is an important core component and it involved remotely located team members. We believed that these settings would allow us to observe more awareness problems caused by the distribution of the development. The CSM component is implemented by a set of features (also known as work items). We tracked the evolution of each feature observing how developers collaborate during its implementation and come up with collaboration patterns. We focused on the analysis of the *Generic Folders* feature, because a broken build was generated during its implementation and it was related to the lack of awareness of changes.

After completing the case study, we have a better understanding of the awareness

problem and its causes. We also identified data source repositories, and finally we developed an approach to address this problem by designing a feature-based awareness mechanism. A feature-based awareness system collects, processes and delivers awareness information from and about contributors in conceptual relationships. A conceptual relationship is a relationship between contributors that are working on the development of a feature.

1.4 Contribution

Our research provides the following contributions:

1. The characterization of awareness problems in a closed-source industrial settings. We document the awareness problems and provide empirical data that supports that the lack of awareness generates problems during the software development process.

2. The architectural design of a feature-based Awareness Mechanism System as a solution to the awareness problem linking contributors with features. The mechanism will notify inter-related developers in a timely manner about changes on artifacts that they are linked to. The awareness mechanism will use what we referred as conceptual relationships among developers to collect and propagate awareness information to the contributors involved in the implementation of a

feature.

1.5 Thesis Organization

The thesis is organized as follows: Chapter 2 includes background material on requirements engineering, awareness in software development, and Social Network Analysis as a method used in our research approach. Chapter 3 describes our research method, including the research questions that were posed during the investigation, and the settings of our case study, the communication tools used by the team members as support of the development, and finally, the data collection methods. In Chapter 4, the findings from the investigation are presented. The discussion and limitations of the research are presented in Chapter 5. In Chapter 6, we present the functional and non-functional requirements and the architecture of a feature-based awareness mechanism system. Finally, in Chapter 7 contributions of the research are reviewed. Also the direction of future research is indicated and concluding remarks draw the thesis to a close.

Chapter 2

Literature Review

In this chapter we review background literature and concepts related to the research goal and approach in this thesis. Considering that a successful development process is based on appropriate requirements change management, we review concepts of Requirements Engineering (RE) practice in Section 2.1. Contributors need to be aware of changes in the software artifacts that they are related to. For this reason in Section 2.2, we present the concept of Awareness in Software Development. This section presents the importance of awareness in workspaces. In Section 2.3, we aim to explore the contribution of previous research in awareness in software development. Further, a solution that involves people, artifacts, communication and other resources are social networks-based solutions. In understanding the ways in which awareness in software development can be maintained, we use the concept of social networks to link

people, who collaborate and need to be alerted about changes in common artifacts. The Social Network Analysis methodology is gaining attention in the development of sociograms to describe collaboration patterns in distributed software development projects. In Section 2.4, we present the Social Network Analysis methodology and its importance to create virtual networks of people, activities and artifacts.

2.1 Requirements Engineering

Computers are present and part of our daily activities. We have learned to trust computer systems as a media to support our common activities. However, using computer systems can sometimes be disappointing. They sometimes fail to work in the way we expected, they provide unreliable information, and they may sometimes be harmful. Thus, they can create more problems than they solve. If computer systems were created to improve our life quality, why do we face problems using them? The answer might rely in the fact that computer systems are *designed* in order to meet a purpose. If our understanding of this purpose is not clear, the design of a system shall not provide the expected outcome.

In order to obtain a good understanding of a system's purpose, Requirements Engineering as a discipline, provides us a framework for understanding the purpose and the context in which the system will be used. RE defines formal processes containing a set of activities to identify the purpose of the system, and to produce a

high-quality reliable product [20] [41] [43]. These activities include requirements elicitation, requirements specification, requirements analysis, requirements validation, and requirements communication.

Despite having well-defined RE processes, one of the keys that defines the system outcome is how well the organization understands and manages their requirements. Not having a good understanding of them could lead to errors. Barry Boehm [3] in the 70's, provided a relative cost to repair a software error in different stages. Errors made in later development phases cost more to fix the longer they are left undetected. A requirement error not found until system testing will cost fifty times more to fix. The rationale is the longer the problem goes unnoticed, the more decisions become based on it, and more rework will be needed to fix dependencies.

We know that the complexity of a computer system implementation addresses the purpose of the system, and the needs of the end-user. If we already know this, and we know how to prevent the failure, why do software projects still fail?[1] There are many causes of failure. May[50] describes how some of them are caused by a poor user input, contributor conflicts, and vague requirements. In their 1994 publishment on software project failures [28], the Standish Group attributed success to requirements related or requirements dependant activities including: user involvement, clear statement of requirements, proper planning, and realistic expectations. Also, the report showed

[1]Cobb's Paradox; Martin Cobb, Treasury Board of Canada Secretariat

that the lack of these factors caused project failures: lack of user input, incomplete requirements, unrealistic expectations, and changing requirements. In 2001 a similar report showed that the success and failure factors have remained the same [29].

2.1.1 Changes: A problem during software development

As reported by the Standish Group, changes in requirements is one of the main factors contributing to software project failure. Change is pervasive in software development processes. During development, all components of a software product undergo change: requirements, design, code, test cases, etc. One major difficulty in managing change is in detecting and propagating the effects of a change to other components of the environment. A change to one component often requires changes to other components that are derived from it or that must be kept consistent with it, and changes to those components must be propagated in turn. Madhavji [48] agrees that the problem of change is defined by a proper impact assessment of a change to other artifacts in the environment in order that all changes are properly made in the artifacts and their interdependences. These changes should record the details for future references.

Sutton [69] also mentions that some causes that originate change problems are the lack of explicit representation of dependency relationships between software artifacts. When an artifact is changed, it is difficult to identify which other artifacts are affected. Consequently, the direction and extent of change propagation is difficult to

determine, and changes may not be propagated completely, and they may be a lack of semantic information associated with artifacts and relationships (rationale). Without some form of specification of changes, it may be difficult to understand the consequences of any given change and therefore difficult to propagate its effects correctly and efficiently.

As an example of changes in software development, we can refer to changes in requirements. Original requirements almost always change after activities of design or implementation have started. These changes are due to external factors (e.g. changing functional requirements, business needs, bug reports, hardware environment, project budget), or internal factors (e.g. refactoring code base or design, bug discovery)[64]. Moreover, determining the impact of changing requirements on a development project is critical to the management of the project [54] and is seen as a fundamental problem in software development [5].

2.1.2 Managing changes in software development

Managing change continues to be a fundamental problem in software development and project management [52]. RE plays an important role in the management of changes in software development [53]. Changes might introduce errors and there is a cost to fix them. Changes in requirements, design documentation, code, test cases, etc. need to be propagated as soon as possible, if the impact on quality and losses in

developers' productivity are to be avoided.

Despite the importance of effective change management, most software development projects do not have a rigorous specification-based requirements engineering process to effectively respond to frequent changes and their impact. Consequently, changes in the requirements specification reside in the system or within the team without reaching relevant contributors involved in the development of the functionality of these requirements in time.

As an example of poor management of changes, research shows that often developers become aware of new or modified requirements long after these changes are included in the requirements repository [17]. In settings that do not include a requirements repository, and the requirements changes are not properly propagated, the errors could be discovered after the system is deployed. Some of the consequences include: developers waste time rewriting code to match the new requirements, (which causes losses to the company), it becomes difficult to identify the responsible parties of the changes and the rationale behind them, and it is not possible to determine the impact of changing requirements on a development project (which is a critical factor in the management of a project). It is also a well known fact that when teams ignore changes in their environment, it often leads to failure in managerial decision-making, software engineering, and new product development teams [6].

Presently, there is a lack of tools and mechanisms to support propagation of

changes during the software development process. As we described earlier, managing changes is an awareness problem, therefore in the next section we present the role of awareness in software development.

2.2 Awareness in Software Development

Awareness has become one of the most important concepts in distributed software development, especially in activities that require support for coordination and communication during software development [39]. The key issue of any collaborative activity is awareness or knowing what is going on [25]. Schmidt writes that the problem with 'awareness' in cooperative work is to understand how actors so effortlessly pick up what is going on around them and make practical sense of it [61]. There is substantial evidence that if teams such as flight crews and software development teams can be aware of the state of the team, tasks and environment they will be more successful [8] [55].

2.2.1 Awareness Definition

According to the Oxford English Dictionary:

> *Awareness is the quality or state of being aware, or the state of consciousness. Another definition for "awareness" is having knowledge or*

cognizance. For example, to be aware of the difference between the two versions; to become aware of a faint sound[2].

However, the definition that better relates to the context of collaboration and the awareness that should be provided in a collaborative work environments is given by Dourish and Belloti [22]; "awareness is an understanding of the activities of others, which provides a context for your own activity." In addition to the Dourish and Belloti definition, the TeamSCOPE research group [67] classified different types of awareness data that are relevant in the effective management of changes and they are as follows.

2.2.2 Type of awareness data

2.2.2.1 Activity Awareness

The knowledge about the projected related activities of other group members is a basic type of awareness information. During real-time collaboration, this may simply mean knowing what actions others are taking at any given moment.

2.2.2.2 Availability awareness

Researchers have learned from system trials that in order for social interaction to take off, people need to decide what kind of interaction is appropriate to involve the target

[2]www.dictionary.com

party. Therefore, knowing the physical availability of your colleagues is necessary but not sufficient. People also need to know "social awareness," such as whether they are busy at the moment, or otherwise unwilling to accept an interaction request despite their presence on the system.

2.2.2.3 Process awareness

Process awareness gives people a sense of where their pieces fit into the whole picture, what the next step is, and what needs to be done to move the process along.

2.2.2.4 Perspective awareness

Anticipation of others actions is important in the coordination of collaborative work. In order to better predict others actions, people not only need information about others past actions, but also information on how particular actions emerged.

2.2.2.5 Environmental awareness

Environmental awareness focuses on events occurring outside of the immediate workspace that may have implications for group activity.

Revisiting the definition provided by Dourish and Belloti, we note that it does not make any distinction between collocated and distributed team members. Both environments contain people and artifacts and they co-habit in a space that Gutwin [30] refers to as a workspace. The complexity to provide and support awareness in a

distributed collaborative workspace is considerably more difficult because one of the primary awareness mechanisms in face-to-face collaboration. For example, observing colleagues' posture and movement is not possible when the team is not collocated [68].

Awareness within a group is useful for coordinating actions, managing coupling, discussing tasks, anticipating others' actions, and finding help [31]. The complexity and interdependency of software systems [44] suggests that group awareness should be necessary for collaborative software development. Team coordination and productivity can be enhanced if people can maintain better awareness of the activities of other teams that may affect their team's work [7]. On the other hand, the lack of information about others' activities can actually harm group morale, such as when team members assume that their colleagues are inactive when they have not heard from them [67].

The following section will discuss awareness in software development workspaces with more details.

2.2.3 Workspace awareness and collaborative software development

A workspace is a space or container that contains all the software artifacts needed by a contributor; any event performed in the workspace necessarily involves them.

Workspace awareness is the collection of up-to-the minute knowledge a person uses to capture another's interaction with the workspace [30]. Gutwin, Roseman and Greenberg [33] pointed out that "workspace awareness reduces the effort needed to coordinate tasks and resources, helps people move between individual and shared activities, provides a context in which to interpret utterances, and allows anticipation of other's actions".

Workspace awareness helps to keep abreast of the changes that occur in different software development workspaces. Although, workspace awareness is taken for granted in face-to-face work, it is difficult to maintain in distributed settings because people only see a fraction of the workspace, and may not see the same part as other group members. Studies of distributed work have shown that much of the communication and implicit information that is available to a co-located team does not exist for remote collaborators [35]. For example, Herbsleb and Grinter [35] found that lack of ad-hoc communication between software developers caused an increase in coordination problems and a decrease in collaboration between remote sites.

In order to address the awareness problem, a collaborative system should support software development and help group members to maintain awareness of each other. The system activities require the capability to monitor actions of interest and alert users when they occur [67]. However, it is difficult to keep track of all events that happen in a workspace during a software development process because it involves having

a monitoring system to record any event where people change artifacts and where these changes affect other people's work in the same workspace or other workspaces that are related.

The purpose of monitoring people's activities in a repository is the need to know "what's going on" in others' workspace(s) that may affect their work. This is especially important for teams that have to work at different times and in different locations [23]. To support awareness in virtual work environments we need a system which gathers awareness information in form of events from the tools used by the developers and distribute them as notifications to interested parties [42]. Despite the advantages that could be provided by a workspace awareness system, the design of it is challenging due to limitations on human attention. This is a major constraint on tools designed to provide passive awareness for distributed work groups. The major problem is that the information needed to maintain awareness of team, task, and environment may overwhelm team members and deflect them from actually doing work [6].

Moreover, in his research, Espinosa [26] obtained some encouraging evidence about the benefits of using awareness tools. He also makes evident how the availability of such tools can be more of a distraction when available but not properly used. For example, a developer does not need to be aware of everything about the project, as the level of awareness depends on the degree to which developers must coordinate,

discuss tasks, anticipate other's actions and find help [32].

2.2.4 Characteristics of an Awareness System

The Palantir project [58] introduces some characteristics that an awareness system should be aligned to. These characteristics are discussed below.

2.2.4.1 Non-obtrusiveness

Developers should not have to change the way they interact with their workspace and tools.

2.2.4.2 Scalability

Informing a developer of all activities in all workspaces overloads their cognitive senses and is not necessary.

2.2.4.3 Flexibility

Not every developer requires the same level of awareness.

2.2.4.4 Configurability

Not always will a developer want to be aware of all activities in all workspaces.

2.2.5 Delivery of awareness data

The delivery of awareness data is very important because it defines how team members will be notified about changes. Previous research [67] provides types of awareness data delivery, and they are as follows.

2.2.5.1 Passive or active delivery

In a passive delivery, the awareness mechanism delivers the information without requiring an action from a user. Active systems, on the other hand, require users to take specific actions to request awareness data.

2.2.5.2 Differentiated or undifferentiated

In a differentiated delivery, the awareness information should reach only users with specific roles based upon their particular expertise or allocated tasks. On the other hand, an undifferentiated delivery of awareness would overload all team members with potential irrelevant information.

2.2.5.3 Customized or fixed

Customization concerns the degree of configurability the users have in determining the awareness information they receive.

2.2.5.4 Awareness information as Focal versus Peripheral

A focal approach directs the group member's attention to the specific awareness data. A peripheral approach delivers awareness data without requiring a user to take his or her attention away from other work.

2.2.5.5 Within a single application or across applications

Group members may receive awareness updates by accessing a single application, or the awareness mechanism may be capable of providing updates to members in a number of separate applications.

In the next Section, we cover some awareness mechanism approaches and then look at similar solutions that have been built specifically to raise awareness in the software development process.

2.3 Awareness Projects

2.3.1 Palantir

Palantir [58] is an awareness mechanism built on top of existing configuration management facilities and concentrates on the collection, distribution, organization and presentation of the relevant workspace information. Palantir increases awareness by *continuously* sharing information regarding operations performed by all developers.

Specifically, it informs a developer of which other developers change artifacts, calculates a simple measure of severity of those changes and graphically displays the information in a configurable and generally non-obtrusive manner.

The main contribution of Palantir is the concept of data-mining events that occur in the CVS repository according to a set of rules defined by the user as a mean to raise awareness among configuration management workspaces. Another major contribution of Palantir is the raise of fundamental questions relevant to the design of any awareness mechanism. These questions are: "Which information should be shared?, and How to avoid overloading developers with information?"

Palantir was used in a case study [58], in which the tool determined the severity of changes and relayed the information to the concerned developers. The three key properties of Palantir are: (1) its coordination mechanism is based on workspace rather than repository information, (2) it continuously instead of sporadically informs developers of other ongoing efforts, and (3) it provides and overall view of other workspaces that supports the detection of both direct and indirect. However, Palantir raises information related only to events originating from the source code stored in the CVS repository ignoring other important events outside the configuration management domain such as changes in requirement repositories, file servers, and bug repositories.

2.3.2 TeamSCOPE

TeamSCOPE (Software for a Collaborative Project Environment) [67] records information on accesses to each team's shared folder and each user's personal directory in a database. This information is used to provide team members awareness of their teammates' activities. Activities tracked by TeamSCOPE include all file-related activities, plus a number of activities related to message boards and calendar events. TeamSCOPE is a seminal piece of CSCW work in awareness. TeamSCOPE researchers developed a theoretical background that later became the foundation of the design and creation of other awareness systems. TeamSCOPE was originally designed as a web-based collaborative system to support distributed teams. However, the contribution of the project goes beyond the presentation and deployment of a web-based tool; researchers defined types of awareness data, mechanisms to deliver information to the end-users, and methods to gather awareness data. Based on the principles stated on the research, TeamSCOPE was built as a tool that provides awareness of activities through a web-based interface where team members are able to share their workspaces in which other group members can store and retrieve shared objects. TeamSCOPE also supports asynchronous group interaction through the ability to post group messages, and provides ongoing information about, the (1) status of groups objects (documents or images) and activities related to these objects (uploads, downloads, or modifications), (2) group communication (log of messages posted in the

message board), (3) resources available to the group (when a resource is available or not available), (4) schedules and availabilities; support group use other external communication resources, interface with other collaborative tools.

TeamSCOPE only tracks changes made through web and ftp interfaces to files stored in a team shared file repository, thus its functionality is limited to document change management. Other software artifact repositories such as CVS, bug repositories, etc., are not considered as a part of the architecture of the system.

2.3.3 The Awareness Monitor

The Awareness Monitor [6] was designed to provide passive awareness of other's activities under the assumption that this will increase coordination and decrease cognitive overload. The project introduces the concept of the development of a passive awareness mechanism system. Passive awareness involves keeping track of events of interest without making a conscious effort to do so.

The most important research contribution is the statement of passive awareness design principles. They are: (1) proportionality, (displays need to be constructed so the important changes in the environment have to be display in the user interface), (2) asynchronous presentation reduces the overload, (3) aggregation, (a display that shows changes slowly and not shows all info at the same time), (4) decomposition, (changes should be displayed according its importance), (5) customizability, (the user

should have the flexibility to personalize the information accordance the importance for him or her), and (6) dampening, (awareness devices should use acknowledge signals for their attention).

According to these design characteristics, the Awareness Monitor was developed and used to monitor and make notifications about activities in a quantitative field study. Students grouped in teams participated in a realistic business simulation called the "Management Game". During the game, teams made decisions regarding a set of tasks in an environment that was constantly changing; each team's decisions affected the other teams. At the end of the Game, students showed a clear preference for being notified of changes to their teams' finances and to the game's environment over information about other's availability or changes to shared documents and file.

There is not evidence that the Awareness Monitor was neither used to support a software development process nor used in software artifacts such as requirements, code, and test cases. The system tracked changes and their availability to shared documents and files stored in a single data repository without considering that this information might be distributed over different repositories.

2.3.4 The Ophelia Project

The Ophelia Project [64] is an initiative that aims to develop a platform to support software engineering in a distributed environment. Central to this aim is the integra-

tion of pluggable, heterogeneous software engineering tools. The Ophelia Project design stresses the importance of relating artifacts to other artifacts using relationships that allow traceability. In supporting traceability, the platform will raise awareness within a multi-site distributed development environment. The project aims to achieve this goal with a four layer architecture: (1) the Kernel Layer, (2) the Tool Modules Layer, (3) the Integration Layer, and (4) the Traceability Layer.

The traceability Layer is considered the most important and is founded on the fact that the artifacts of the software engineering process are represented as CORBA objects. Having all artifacts of the project, Ophelia captures all the relationships present and not only those derived from requirements, for example, but also relationships between defects and code, and defects and models, etc. Having an abstract uniform interface to all elements of the project, Ophelia generates an explicit graphs of relationships among these elements providing an opportunity to implement automatic notifications about changes occurring on them.

The Ophelia Project provides only the architecture of the system; there is not evidence neither of its implementation nor validation. Another important observation is that artifacts are related to each other without a rationale of the relationship, and the architecture does not consider links between artifacts and contributors.

2.3.5 The Jazz Project

Jazz [12] is an Eclipse extension developed by IBM. It is based on the metaphor of an "open office" approach to application development. In such a setting, a team of developers works in close proximity. Within the office each developers will have his own workstation, while elsewhere in the office will be space for team meetings, shared whiteboards, and schedule information.

The contribution of the Jazz project is in terms of providing synchronous awareness about changes on code performed in the CVS repository, Jazz is built as an extension of the Eclipse IDE, which means that is an awareness mechanism integrated with a development tool, which reduces the need to use another application(s). Jazz also aims to provide the user peripheral awareness of the other team members. It shows the user and lists the teams the user belongs to, as well as the members of these teams. It also provides easy and in-context access to as much of the team information as possible, without hindering the user's work unnecessarily.

So far, Jazz only supports code-change (CVS) awareness. However, Jazz has become the foundation of current development of collaborative support applications, and its evolution includes enhanced functionality. For example, Jazz has its own Source Code Management (SCM) and bug tracking system. Having this new capability, Jazz users will be able to create their own workspaces and synchronize their own artifacts with a central repository. The new capabilities developed in Jazz will make

it a powerful tool to support distributed development and deliver awareness among the members of a team.

2.3.6 The EGRET Project

EGRET (Eclipse-based Global Requirements Tool) [63] is another Eclipse extension developed by IBM. This tool uses a database back-end repository and CVS to control changes to the requirements repository. The tool supports synchronous and asynchronous "contextual collaboration" around requirements. The idea is to provide a shared context to contributors separated by distance, in order to make discussions among them smoother and transparent. In the EGRET project, researchers conducted a field study in which after interviewing project members located in USA, Netherlands, and India, they identified challenges related to distributed requirements management.

The most relevant contribution of this research is the design of a tool which will (1) support informal collaboration services that facilitate ad-hoc conversation around requirements, (2) support critical/regular processes (i.e. change management) in requirements phase and reduce the need for human interaction, (3) provide awareness about pending changes, and (4) support knowledge management to navigate project content. EGRET was evaluated by 12 practitioners from 3 different projects (all based in India). The overall evaluation of the tool was positive; practitioners found the

tool very useful. However, some performance issues were raised during synchronous communication.

So far, EGRET only supports the communication of requirement changes across distributed teams. However, a change to a requirement is only notified to the owner of the requirement leaving other contributors unaware of the change. Moreover, other artifacts such as code, design documents, and test cases, are not currently consider in the meta-model of the tool.

2.3.7 Ariadne

Ariadne [70] is a Java plug-in for Eclipse. Ariadne analyzes a Java project to identify program dependencies and collects authorship information about the project connecting to a configuration management repository. By extracting dependencies in the source-code, Ariadne generates a call-graph, in which displays the dependencies among software developers responsible for the implementation of software components. The social call-graphs describe both technical and authorship information, they can be used to generate sociograms describing the dependence relationship only among software developers, that is, dependencies between social developers because of dependencies in the source-code they are working on.

Ariadne only extracts dependencies based on code ownerships from the CVS repository and the configuration management system. Other software artifact repositories

such as bug and communication repositories, are not considered as a part of the architecture of the system.

We presented in this section different approaches from the academia and industry that tackle the awareness problem. However, none of these approaches tackle completely the problem of linking all software components with the contributors involved in their implementation. Therefore, we can conclude that we still need a system that will link people with artifacts, and in particular requirements. Current tools do not support this need, and we believe this is one way to maintain awareness about changes in the requirements and subsequent artifacts linked to them.

In the next section, we will review the concepts of Social Network Analysis as a main methodology in our approach to link people, processes and artifacts in software development.

2.4 Social Network Analysis

A social network is a social structure made of nodes which are generally individuals or organizations. It indicates the ways in which they are connected trough various social familiarities ranging from casual acquaintance to close familial bonds [2]. The social network paradigm is gaining recognition and standing in the social and behavioral science communities as the theoretical basis for examining social structures [71].

> *Social Network Analysis (SNA) seeks to describe patterns of relationships among actors, to analyze the structure of these patterns and discover what their effects are on people and organizations* [49].

The idea of examining the structure of social networks has been adopted by a wide variety of researchers. Social network techniques have been around for some time as valuable tool in the fields of Psychology and Anthropology [16] to create "sociograms" to connect people and create social structures based on the concept of roles and positions. For example, sociologists use network analysis techniques to examine the migration to urban environments on the composition and resources resident within social networks [72]. In mathematics, Graph Theory provides analytical techniques to harvest information from a social network since few decades ago [34].

Social networks cope with the structural problems of the organization, work processes, geographic dispersion, human resources practices, leadership style, and culture. These organizational problems become more complex in knowledge-intensive settings [15] such as the software development practice, where project managers count on the collaboration among developers with different expertise. Developers rely on their network to find information about projects components, infrastructure and experts that can help to solve problems that they face everyday [14].

Some examples of how SNA techniques support software projects are the research performed by Gonzales-Barahona and Robles [27], in which relationships were data-

mined from a CVS repository in order to create structures of the Apache Project community. Adamic and Adar [1] defined the structure of the corporate social networks analyzing email logs at HP Labs. Cho [13] used SNA features to identify and understand communication and interaction patterns when collaborating through wireless computer networking tools. Souza [66][70] argues that the source-code itself can be an important resource to identify social dependencies, and can help in the creation of social relationships among software developers to facilitate the integration process.

In our research, as described in the subsequent chapters, we observed a group of software developers to determine the development of communication networks in order to propagate changes on features, and to identify the developers that are linked to the development of a particular feature. We believe that having a good picture of the information contained within feature-based networks [3] could help in tackling the awareness problem in software development projects, because networks plot the relationships among contributors that need to be alerted of changes in features.

2.4.1 Conducting a Social Network Analysis

SNA makes visible collaboration patterns. Plotting them will help managers and researchers to visualize the team members involved in the development of particular

[3] A feature-based network is a group of people working on the same feature. The network includes people who are explicitly assigned to the features, and people who become involve in its development.

features. In order to create this feature-based networks, a set of steps must be followed to obtain the expected results.

- First, researchers need to know what the goals of the research are, and what information needs to be displayed in a sociogram. For example, what are the communication patterns between developers during the implementation of a specific component?, what communication media support the development?, who are the developers that handle a high volume of communication? When, the SNA research questions are clearly defined, we will move to the second step.

- In the second step, the researcher needs to define the size of the team that will be observed and data collection methodology. The number of people is very important, the larger the number of participants a more accurate information will be obtained. The data collection methods need to be aligned to the research questions and the accessibility to the participants. Data collection methods such as surveys and interviews are most commonly used and easy to apply. In order to draw a sociogram that plots communication patterns, the researcher will need to collect information about a particular feature such as name of the contributors that implement the feature, name of contributors that they interact with, communication media used during the interaction, reason for the interaction, and date of the interaction.

- The third step, once the data has been collected; researchers can draw sociograms that will represent the relationships among team members. The generation of sociograms might be either manual or automated using specialized software packages (These can be obtained from the International Network for Social Network Analysis - INSNA[4]). Similarly, network analysis software is available that can derive metrics from the data; for example it can show through quantitative analysis who the central people are in a given network [16]. From the socio diagrams we can visualize important data such as the volume of communication that they exchange with other team members, and the media that they use.

2.4.2 Example of a network diagram

The information collected from social network surveys or case studies can be used to create network diagrams that illustrate relationships between members of a group. For example Figure 2.1 shows the flow of information within a globally dispersed software development team. It is composed of subgroups of people linked by lines and arrows that show how information is propagated between group members. According the volume of information hosted by the network, central and peripheral people can be easily identified. The following describes more details about each component of a

[4]INSNA Website: http://www.insna.org

network diagram.

- **Subgroups:** Groups within a group often arise according the needs of the project, in Figure 2.1 we observe that the development has been split over 4 groups that are geographically distributed.

- **Peripheral People:** Some people are only loosely connected to a network, a few may be completely isolated - members in theory but not in practice. In the diagram for example, Scott, Tyler, Xavier and others are peripheral people.

- **Lines and arrows:** Each line indicates an information link between two people. Arrows represent the direction of the relationship (incoming arrows show that the person is a source of information; outgoing, that the team member seeks information from the linked party).

- **Central people:** Network diagrams make clear who the most prominent persons within a group are. In the diagram Zoe can be considered a central person.

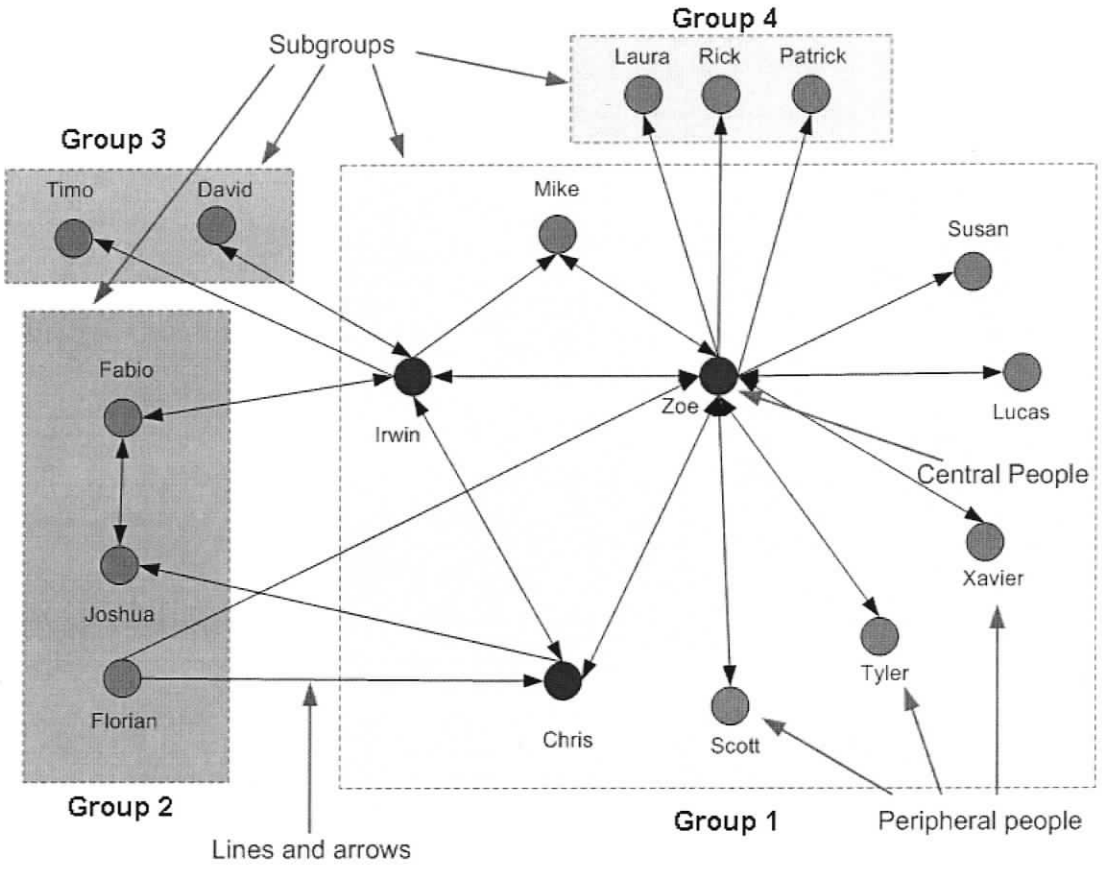

Figure 2.1: Social network components

Networks evolve and grow in parallel to the evolution of the development of features, and this evolution also implies continuous changes in the number of people conceptually linked to a particular feature. Hence, developers need to know the extension of their network in order to evaluate the dependencies of their work, and be aware of the number of people that need to be alerted about changes that they perform in their software development workspaces.

Developers use networks to seek out expertise in finding out the rationale of changes in features. Considering that the concept of "rationale of changes" is related to the description of a requirement, we can say that developers use networks to leverage expertise in order to have a better understanding of the requirements of the system.

2.5 Summary

This chapter has covered the importance of Requirements Engineering, and the management of changes in a software development process. The literature suggests that a good change management practice can avoid project failures. We have also covered the role of awareness in software development, and the importance of keeping contributors alerted about changes in their workspaces. Tools that try to cope with the awareness problem were described; however, we also identified the need for a feature-based awareness tool to support the complete software development process.

Finally, the Social Network Analysis was considered as our approach methodology to determine the relationships between people and artifacts.

The following chapter presents our research method and case study design.

Chapter 3

Research Method

In this chapter we present a case study conducted in one of the IBM Research Centres in Canada. IBM allowed us to observe a multi-site development process of one of its applications under development. We define our research questions in Section 3.1. The complete case study design is presented in Section 3.2. The design includes description of the company, the project under development, and the development process. In Section 3.3, we describe the synchronous and asynchronous communication tools used to support the development process of the system. Finally, in Section 3.4, the data collection and analysis methods used in our case study are described.

3.1 Research Questions

The lack of awareness is one of the main causes of problems related to the propagation of changes; this problem is extensively discussed and presented in the previous chapter. In order to gain a better understanding and characterize awareness problems, we conducted an exploratory case study in an industrial setting. A set of research questions guided our observations:

Q1: How are developers alerted of changes during the development of a feature and which communication media are primarily used to propagate changes?

Rationale: This is probably one of the most sensitive questions. Answering it will help us to understand how developers are alerted of changes, and which communication tools are more prone to be used to propagate information among concerned developers.

Q2: Which awareness problems arise in distributed development settings?

Rationale: Our aim in answering this question is to determine if geographical distance and other factors related to physical distribution of team members play an important role in the propagation of changes within the developers network.

Q3: What are the collaboration patterns relevant in the study of awareness in feature management; i.e: roles and interdependencies be-

tween people and features?

Rationale: The rationale behind this question is to determine the collaboration patterns that will be affecting and/or supported by an awareness mechanism. This informs the level of involvement of a developer in the development of a features of the system and how the communication of changes is propagated among developers involved in the implementation of the same feature. Specifically, answering this question will help us to understand how a developer collaborates with other members of the team, and how tools should support communication during the software development process.

In order to answer the research questions, we have designed a case study of a project and its development to collect data in an attempt to answer these questions. The following section presents the details of our case study.

3.2 Case Study Design

From July to December 2005, I was located at the IBM Ottawa Software Lab, and observed the daily interaction within a distributed team with locations in United States and Europe. The information gathered during this period of time was obtained using: interviews, document inspection, Bugzilla repository inspection, mailing list inspection, activities log compilation, social networks analysis, monitoring of the build's health status, and observation of team meetings. The following will detail our case

study setting.

3.2.1 Study Setting

3.2.1.1 The Company and the location of the study

Geographically distributed software development is pervasive among most large companies [35] including IBM. We chose one division of the company for this study. This division is involved in a large project that requires extensive coordination due to its multi-site development work. We decided to study this division for the following reasons:

1. The project was willing to host researchers and provide us with access to developers, documents, and other resources.

2. The division engages a number of cross-site collaboration, with other IBM sites in North America and Europe.

At the beginning of the case study we wanted to observe people linked to requirements; however, we realized that due to the lack of formal Requirements Engineering practices, requirements were actually being called features or work items[1]. We were also interested in exploring the coordination required in the development of certain

[1]Please note that in our document, every time that we mention a work item, it should be consider as a feature.

features, identifying the tools used to support this process, and identifying how awareness of changes at the work item level is maintained in the project.

3.2.1.2 The Team Tools Project

Team Tools is the generic name of an IBM's multi-site distributed software development project that we observed. The aim of Team Tools is to improve the performance of the Eclipse IDE providing a multi-distributed development capability features. Another feature of Team Tools is that it provides better support to large-scale development projects. Scalability is one of its major design concerns because this feature is not well supported by similar off-the-shelf products available in the market.

The development of Team Tools has been split among teams with a total of 60 contributors remotely located in 12 different locations across North America (Canada, and USA) and Europe (Switzerland). Each team contributes to the project in design, implementation and, testing and support; although only 4 teams out of 12 are in charge of the implementation of the core components of Team Tools:

1. The User Interface (UI) Foundation Team - Europe
2. The Team Central - USA (Location 1)
3. The Server Team - USA (Location 2)
4. The file content server team - Canada - USA (Location 3)

Modularity is one of the major characteristics of the Team Tools architecture. The components' implementation is distributed among the teams described in the previous section. I had a firsthand experience with the development process of the File Content Server components (located in Canada). The following will provide a description of the development process of the File Content Server components.

3.2.1.3 Development process for the File Content Server

The File Content Server provides two major components: component 1 (CSM), and the component 2 (User Interface).

Team Structure (Members and roles): The team has two subgroups, the first one is the CSM core team comprised of one team leader and other 3 developers. The second one is the UI team comprised of 3 developers. The UI team relies on the implementation of the core CSM APIs. One extra developer provides technical support in the use of Eclipse Modeling Framework (EMF). He is also in charge of the integration of CSM with the central repository (based in USA - Location 2). The team leader is a floating developer, he supports all the development and handles coordination with remote teams. An extra floating developer supports either the CSM and the UI teams according the needs of the project.

Discussion and implementation of features: The File Content Server component has a set of features that are implemented for the team. The following describes

the process of implementing a feature:

1. Team leader (TL) coordinates which features would be feasible to implement for the next milestone with other team leaders and system architects.

2. TL comes up with a list of tasks that will implement the desired features.

3. In a session meeting the TL discusses with developers about the current stage of the development. Meetings start with the presentation of the advances of previously assigned tasks.

4. The team members have the opportunity to discuss about the presentation and the problems that they found during the implementation.

5. Developers' ideas provide guidelines about how to fix found problems; some architectural issues are also discussed during the meeting.

6. TL proceeds with the distribution of tasks; a list of tasks that have to be performed for the next weeks is shown to developers.

7. Part of the functionality comes from ideas sparked during the meeting, under the phrase "It would be nice to have this as a part of the application."

8. The tasks that are assigned to a developer do not contain any details; it is up to the developer to choose how to implement them.

9. Developers meet individually with the team leader or other developers to discuss the implementation details. These meetings are very informal. They walk to each other's offices, or they discuss using SameTime [2].

10. Developers implement the functionality and write a set of test cases. When the implementation is finished, they communicate to the TL that the functionality has been implemented and ready to be tested.

11. TL provides feedback about the implementation.

12. Developers improve first iteration based in feedback.

13. After improvement, developers commit their changes to the central repository, and communicate to team members that the functionality implemented is ready to use.

14. Developers present their implementation to the team in a "deep dive" session.

From the steps described, we observed that the development process is very Agile in terms of managing the implementation of a feature from inception until its deployment in the CVS repository. We could also note the lack of formal Requirements Engineering practice. In fact, we observed that the rationale behind the implementation of a feature follows an OSS management style [60]. The success of a correct implementation relied on the following factors:

[2] An IBM IRC tool part of the Lotus Notes Suite

1. Experience and expertise of application architects and team leaders.

2. Team Tools is a product that will be used for developers and developers are implementing it. Thus, architects perceive that formal specifications or more details are not necessary due to the fact that developers already know how to implement based on their experiences and their wishes about how they would like to use the application.

The developers' knowledge of the architecture has evolved in parallel to the growth of the functionality of the system. The advantage in this case is that developers have a good understanding of the impact of their changes in other workspaces. Thus, when they are asked to develop a specific feature they know what they have to do, what is possible, and the potential effects of their changes in others' work. However, the system keeps growing in functionality and complexity, making it more difficult for developers to keep abreast with all the details that involve changes in their workspaces. Therefore, we believe that regardless of the developers' expertise, they will need to use a mechanism to keep them alerted about changes in their workspaces. The following section will provide more insights of the development of the CSM component.

3.2.1.4 The CSM Team

We focused in the observation of the development of the CSM component for the following reasons:

1. It is an important component of the project.

2. There are clear goals defined for the incoming milestone.

3. Part of the local team (Ottawa) will be involved in the implementation.

4. Its development requires active communication with a remote site (USA - Location 3).

In this section, we will introduce the background of the two teams involved in the implementation of features of the CSM component. The background of the teams will help us to understand our research findings presented in Chapter 4.

About the Ottawa-based team before involvement in this project

The Ottawa-based CSM team members have different backgrounds. The team leader of the project belongs originally to Object Technology International (OTI), and he has been one of the main Eclipse API architects, with extensive experience in CSM API design. The other core members of the team originally belonged to Rational, and they were involved in the development of the ClearCase's File Server system.

About USA Location 3 team before involvement in this project

The USA-Location 3 IBM Centre was one of the branches of Rational; ClearCase[3] was created in this Research Centre. In parallel to the development of ClearCase

[3]ClearCase is a tool for revision control (configuration management, SCM, etc.) of source code and other software development assets.

UCM 2 (Unified Change Management), part of this team started the development of an application to support distributed development teams when not even IBM had the conception of this type of applications. They used their experience in ClearCase to start modeling and implementing a central Source Code Management repository. The implementation of this project was not the priority of the team; however, the research and time spent in the development of this functionality provided an advantage over other teams and companies that did not envision the potential of this project in the past.

CSM project planning

We observed a lack of formal Requirement Engineering and Project Management practices. We also observed that the original plan of the CSM was designed for the CSM's team leader and posted in the project WIKI. The planning document included a set of work items, which were prioritized according to their importance and dependency or other work items.

For each work item, a bug ticket was created in Bugzilla, where all the discussions were hosted. Some of the work items were carried from the previous milestone; hence, some of them already had some design notes that were improved before starting the implementation.

In this section we have described the settings of our case study, in the following section, we describe the data communication tools used during development.

3.3 Communication tools used during development

Team Tools used a variety of synchronous and asynchronous communications tools to host the interaction of developers. The tools used were: WIKI, mailing lists, meetings, SameTime and Bugzilla. We now provide a brief reference of each of them.

3.3.1 Asynchronous communication tools

- **WIKI:** The project WIKI is used as a documents repository, in which developers post information related to the implementation of features. The WIKI contains planning, design, and common interest documentation. Developers have to login to the WIKI in order to retrieve and post any document in it.

- **Mailing Lists:** All the Team Tools members are members of the project mailing list by default; therefore, they receive all the emails sent to it. The mailing list is used as a notification media to propagate common interest information such as build generation notifications, or locks in main design components. Other mailing lists are created according the need of the teams; for example the CSM team has its own mailing list in which only messages related to the implementation of the CSM components are sent; needless to say that only the CSM team members are the receivers of these emails.

- **Bugzilla:** As explained in Section 3.2.1.4, Bugzilla's notification system is used to host discussion around a feature. Only developers involved in the discussion of this particular "bug" are notified about a new posting in Bugzilla.

3.3.2 Synchronous communication tools

- **Meetings:** There are two types of meetings: deep dive and project meetings. In the deep dive meetings, teams from several locations meet using share screen and audio conference systems to navigate the current functionality of the system. In the project meetings, the members of the local team meet in person to go over the planning and distribution of tasks. It is important to mention that developers held informal face-to-face meetings to discussed different issues related to the system implementation.

- **SameTime:** Is an IRC tool part of the Lotus Notes suite widely used during development, specially to communicate with remote peers.

- **Phone:** This tool is primarily used to reach remote peers and tackle problems that demand immediate solution. Although, this tool was available for anybody in the project, it was more commonly used by managers and team leaders to coordinate and plan implementation.

In the following section, we will describe the data collection methods used in our case study.

3.4 Data collection methods

The data collection methods used in our case study are the following.

3.4.1 Interviews

We interviewed a total of five team members, two of them were team leaders and the other three were developers. The objective of interviewing the team leader was to obtain a better understanding of the development strategy and to choose a multi-site development component that requires high interaction with remote located team members. A core component, which we will call CSM from now and on, was identified as appropriate for our case study. We interviewed the team leader of the CSM team. The objective of this interview was to understand the component development complexity, identify developers involved, define the data collection strategy, and feature development timelines.

During 19 business days, short daily meetings (between 5 to 10 minutes) were held individually with the CSM team leader and other three developers involved in the development of the CSM component. The objective of these meetings was to

obtain firsthand information about daily communication activities. The information compiled was included in the activities log report. We provide more details of the activities log report further in this section.

3.4.2 Observation of team meetings

The researcher was a non-participant observer in two deep dive and one project meetings. We observed and documented how feature implementation details are communicated from team leaders to developers.

3.4.3 Document Inspection

We inspected the documents related to the development of the CSM component posted in the WIKI. Each posting was recorded in a spreadsheet. For each record we recorded the date, name of the document, the person who posted, and the reason of posting. At the end of the data collection period we summarized the total number of postings. Most of the project documentation generated for the contributors is posted in a project WIKI. The WIKI contains modules for documentation related to design, planning, support, and other miscellaneous items. We could not determine the exact number of pages, but we can estimate over 200 hundred pages in total. The documentation inspected contained planning layouts, and design documents. We aimed to identify the volume of information generated for each feature included in the CSM

planning document that could be relevant as an input for an awareness mechanism system. Postings to the WIKI are not considered communication interactions because the WIKI was used as documentation repository and project memory.

3.4.4 Bugzilla Repository Inspection

In the CSM component planning document, a set of features was listed. For each feature a bug ticket was opened in Bugzilla, where developers posted comments about the "bugs" (features), and received automatic notifications of new postings. We followed the discussion for each feature defined in Bugzilla during 19 business days. Bugzilla was used in the project not only as a bug tracking system, but also as a the notification mailing functionality is to host the discussion of features. Each posting by "bug" was recorded in a spreadsheet; each record contains the date, number of the bug, name of the poster, and the reason of posting. At the end of the data collection period we summarized the total number of postings per bug.

3.4.5 Mailing list inspections

The CSM team has a mailing list to host communication related only to the implementation of the component. The messages sent to the list were inspected in order to identify and classify the issues discussed by the project members, and the frequency that these issues were discussed. In order to obtain a proper classification, we looked

in every email for keywords such as sync-up, change, planning, coordination, support, risk, and implementation. We counted the occur. If messages included more than one keyword, we analyzed the context of the email to come up with a proper classification in just one of the categories.

3.4.6 Developer diaries

The CSM team members were invited to participate in the research. They were asked to keep a log of their daily communication activities with members involved in the implementation of CSM components. Four developers located in Ottawa agreed to participate and provided a log of activities during 19 business days. Two developers located in United States also agreed to participate and they submitted their communication logs during 5 business days.

The activities log contains information about the feature that developers discussed. The logs included the contact person(s), the reason(s) of the communication and the media used. Table 3.1 shows an example of the information compiled.

Table 3.1: Dairy kept by Irwin

Date: Nov 9th 2005			
Team member: Irwin			
Location: Ottawa			
Feature	**Contact person**	**Reason of Communication**	**Media Used**
Bug 0001	Trish	Update of the module1 development	email
	Fabio	Coordination of the implementation	phone
Bug 0002	Florian	Discussion and comments about the new proposal	chat

From the activities log compiled, we extracted developers interactions per feature; for example, all the interactions for the Bug 0001 (see Table 3.1), classified them according to the communication tool used during the interaction, and we drew a sociogram per feature (see Figure 4.4 in page 78). To develop a socio diagram we used a light version of the Social Network Analysis (SNA) methodology to create a visualization that includes developers, volume of communication, and communication media used during the implementation of a feature.

3.4.7 Monitoring the build's health status

We daily monitored the status of continuous and integration builds that contained the CSM components during a period of 19 business days. We wanted to identify the existence of broken builds, and identify whether they were caused by the lack of awareness or other reasons. Our assumption was that if developers knew whom they would be affect during the implementation of a feature, they could notify or be notified about changes that would affect their workspace and thus avoid broken builds. When a broken build was generated, we recorded the event and asked the people responsible for the generation of the build the cause of the failure.

3.5 Summary

In this chapter, we have presented the research method used in our investigation. The research questions that guided our research were introduced in the first part of this chapter, and as a main research method we have presented our case study. In describing the case study design we described the company, the project, and the specific component that we observed. We have presented the communication tools used by the team members as a media of propagation of changes. Finally, we presented the data collection and analysis methods used to gather information in the project in order to answers our research questions. In the following chapter we will present the findings of our case study.

Chapter 4

Findings

In this chapter we present the findings from our case study. Findings are categorized in two main sections. Section 4.1 describes the use of communication media and classifies them according to the volume of information handled. In Section 4.2, we present the two main reasons that cause awareness problems. In section 4.2.1, we present an analysis of the communication logs as a means to provide evidence of how corporate culture issues have impacted on awareness during the development process. In Section 4.2.2, we present a case where we found a link between the volume of communication and the generation of broken builds.

4.1 Communication media use

During a period of 19 business days, a total of four members of the CSM Ottawa team were asked to keep a log of communication (See table 3.1 in page 58) activities with local and remote members in the development of work items of the CSM component. Two developers located in US submitted the communication logs via email. The criterion for the data compilation was the same for local and remote teams, as described in Section 3.4.

The names used in this document are fictitious to protect the participants identity. We will refer to Irwin, Zoe, Chris, and Lucas (see Figure 4.2 in page 74) as members of the CSM Ottawa team, and Trish and Joshua (see Figure 4.2 in page 74) as members of the US team.

From the compiled logs of communication, we counted a total of 418 interactions among CSM developers during this period of time. If the same interaction was reported for more than one developer, we just counted one. Table 4.1 contains the number of interactions as reported by the CSM developers and the number of postings that they performed on Bugzilla.

Table 4.1: Number of interactions reported and postings on Bugzilla

Name of developer	**Number of interactions reported in the questionnaire**	**Number of postings in Bugzilla**
Trish	34	9
Zoe	58	6
Joshua	23	2
Irwin	92	20
Chris	33	18
Lucas	5	0

Figure 4.1 displays the total number of times that developers reported using communication tools. As shown in the chart the highest number of interaction of the developers took place during "face-to-face" (F2F) meetings; a total of 119 out of 418 (28%). Also, 28 out of 418 interactions were using the phone (6%). This means that if both percentages are added (F2F and phone), the 34% of the total communication among the team members is neither recorded nor stored.

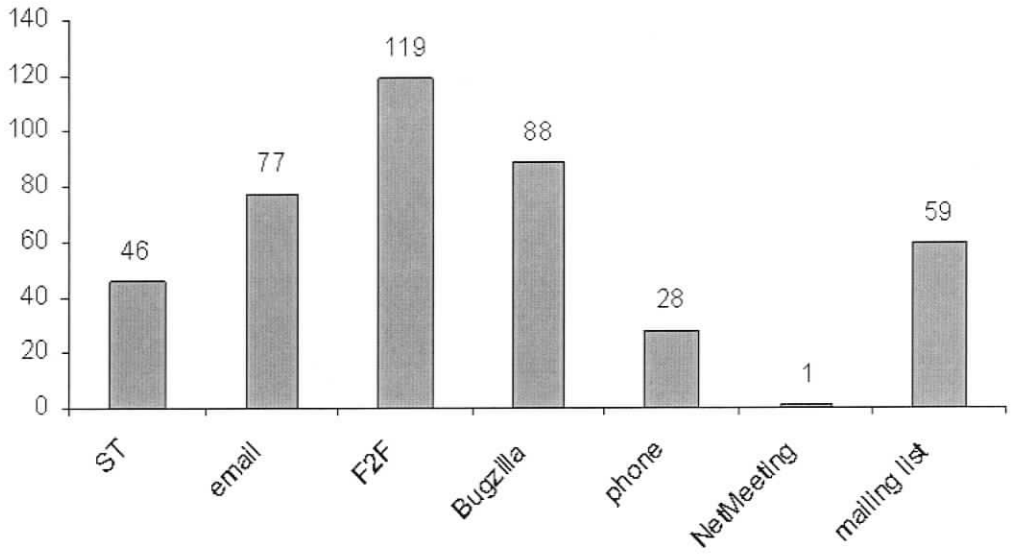

Figure 4.1: Number of reported interactions per communication media

Another interesting finding that is particular for this project is the high number of interactions using Bugzilla. As briefly described in the previous chapter, Bugzilla was used to host discussion about the design proposals and implementation. As shown in Figure 4.1, 88 postings were submitted to Bugzilla. This means 88 interactions

of developers with the team out of 418; thus, 21% of the total communication was hosted in Bugzilla. The other communication media used for the team members were email with 18%, SameTime(ST) with 11%. The use of NetMeeting did not represent a significative percentage of the total amount of use of communication tools.

4.2 Awareness Problems

4.2.1 Lack of awareness: Corporate global cultural issues

In large companies such as IBM, there is a diversity of products, teams and technologies. Teams develop their own culture that includes work-style and communication-style. In our case study we observed how team culture plays a very important role in how awareness is raised to alert remote located developers about changes in work items, features and components that are cooperatively implemented.

Analyzing email communication logs between non-collocated team members, we identified that miscommunication and lack of common understanding raised serious problems, such as schedule delays, lowered level of trust and slowed integration processes of a new team into development. One of the main concerns of the Team Tools management is to have a process that allows new team members to get involved in the project without creating an overhead. In our observation of the interaction between an experienced Team Tools development team (Ottawa) and a new team in

the project (US Location 3), developers at one site assumed that their remote peers always referred to the build notes to review the list of changes. The other team relied on communication protocols using a more conventional media such as email to propagate changes regardless of their importance. This lack of knowledge about how to propagate changes harmed the trust relationship between teams, and slowed down the integration process of a new team into development.

To illustrate these observations, the following messages are included from a thread posted in the CSM mailing list; the discussion provides some insights about a particular case where developers from both teams found a problem caused by the lack of awareness of changes. One of the developers in the remote site (referred to as US_Dev1 henceforth) found that his work was affected due to changes made by Ottawa_Dev1, US_Dev1 was not alerted of this change, and so was his reaction:

> *US_Dev1 : "When I came in this morning, expecting to be able to deliver after last evening's work, I was thwarted by changes that I wasn't expecting. That pushed back my schedule, in turn pushing back US_Dev2's schedule."*

As the example shows, one of the effects of the lack of communication is the delay of schedules and frustration generated on the developers affected. The concerns of this developer goes further when he notes that it has not been any discussion about how to communicate the introduction of new classes:

"The second issue is the introduction of new classes that I didn't hear any discussion on. Maybe I missed it. Or maybe the classes were deemed too trivial to comment on - that's fine. However, ***in the culture I work in,*** *folks send out mail on new classes. Sure, I can see it in the sync report, but I think it helps to get a head's up on it, esp. when we're not sitting in the same office."*

On this part of the message a developer points to the fact that due to the team distribution, they can not rely only on the sync reports, it means that they would prefer to use a mechanism that alert them about the changes, regardless if a log of changes is generated during the build creation. Another interesting observation from the analysis of this portion of the message is that the developer explicitly mentions "the culture I work". He is fully aware of difference of culture and practices in both teams. This action generates a reaction in the other site (Ottawa). The response from some members in the local team to the implicit proposition of using email as a primary communication tool to propagate changes at different levels of granularity was:

*Ottawa_Dev2: "****To me, emailing every intention/commit is overkill.*** *Is there not somewhere in between that we can meet?"*

The answer to the proposal is a natural reaction of any person that is concerned

about the volume of communication that could be handled. The concern of this developer is one of the major challenges of the design of effective awareness mechanisms. Besides the concern of the volume of information, the idea of using email as a primary source of awareness and propagation of changes is not necessarily welcomed for some developers for reason that are not necessarily related to development. In one of our daily interviews, one of the developers mentioned that he did not like to use email because of the perception of leaving a copy of the communication in the email server of the company, which could be used to measure his performance. Team members in both sites agreed that the use of email to communicate any minor change is overkill, as it is shown in the following exhibit:

> *US_Dev2: "I sympathize with your feeling that "check-in email" for every minor change is overkill. I also think that given the newness of our distributed team, we need to err on the side of over communication for the time being."*

On this part of the message this developer also recognized that the use of email would be overkill, however, there was still a need for distributed teams to exchange extensive communication especially if the teams had not worked together before. Email was not necessarily the best or the only option available. The following exhibit shows how the same developer proposed another solution using the project's settings and technology.

"Is there a way for the CVS server to be configured to send check-in email, such that if we put useful comments into the commit dialog box, others would see it? CVS has such a feature "

Developers on both sites agreed that the use of email could harm productivity. One of the concerns was "I should spend more time coding than communicating." However they were aware that they had to reach a balance between amount of communication and other activities, and also that the technology played a important role in how communication would be held among team members. However, CVS was not the only program that provided notification functionality; one of the developers in Ottawa brought up the idea of using Bugzilla, since this tool was used to propagate changes at the work item level.

Ottawa_Dev2: "Point taken. Rather than explicit emails, I would be happier with updating the bugzilla entries, so to consolidate discussion and to avoid interaction with lotus notes."

Discussion emails sent back and forth between team members trying to reach an agreement finally paid off; members agreed to use Bugzilla notification functionality as a system to keep members alerted about changes in the implementation of work items.

This discussion illustrates how issues are often raised and discussed over email, and how developers come up with possible solutions that can be beneficial for themselves.

Observing the discussion on the mailing list provided us with a good understanding of communication problems that developers are coping with. We realized that the discussion demonstrated the low level of trust between the developers on both sites.

Despite all these challenges, team members were still willing to work out possible solutions to overcome barriers created by distance and corporate cultural factors.

4.2.2 Lack of awareness: Broken builds due to the high volume of information

We were able to identify the generation of a broken build caused by an integration build by non-timely communication between team members. The generation of the weekly-integration build happened every Monday evening. Before the automatic generation takes place, the developer responsible for the integration build's preparation sent a notification email to the CSM developers to keep the code stable throughout the day. Due to the amount of information handled by Zoe (see Figure 4.4 in page 78), attending this notification was not possible for her; in fact she did not have a chance to read the notification email at all. When the generation of the integration build occurred, the person responsible for the build's creation noticed that the failure of the build originated from changes made by Zoe. Changes were rolled back to the point where the generation of a green build was possible, resulting in a loss of time to recover and generate a build that was not meant to be broken.

We will soon describe with more detail the evolution of the communication in the work item that generated the broken build, but first, we will describe how a build is generated.

4.2.2.1 How is a build generated?

The generation of continuous and integration builds in the project allows project team members to monitor the system's health and keep working with the latest stable build (also called "green build"). Also, continuous builds alert developers of the failures produced by commits in the repository (failures produce "red builds"). The build is created automatically for an application called Cruise Control[1], which fetches all the files that will be part of the build, and creates a map file that will load all the system components from the locations specified in the map file. The map file contains the location of the files. These can be called from the different branches of the CVS repository. The CVS repository of the project is divided into branches. The two main branches are HEAD and STABLE. The content of both is similar, the main difference being that they have different versions of the artifacts. The HEAD branch is always ahead of the STABLE branch; however, the build can retrieve files from both branches at the same time. If the application that generates the build catches a missing reference, it automatically triggers an exception and the build is not generated

[1]Cruise Control is an Open Source wrapper program for automating builds developed for Java.

(changing the status of the build to a "red build"). We observed that the two main causes of the red build were:

1. The lack of understanding about the structure of the build (which components should be extracted from HEAD and which should be from STABLE).

2. Non-timely communication concerning changes prior the generation of the build.

The first cause of the red build was related to the definition of business rules about the components that were part of the build. The aim of the research was to identify factors that were related to the lack of communication or coordination. We provide more details about the second cause in the following section.

4.2.2.2 Work Item Evolution and Social Network Analysis

We believe that the high volume of information received by developers during a typical day makes them prone to skip or miss important communication that could cause problems such as the generation of a red build. We observed the evolution of 12 work items of the CSM component in order to explain our assumption. However, we presented only the details of the evolution of communication patterns and development of the work item *Generic Folders* (bug 1009). This particular work item is one of the core features of the CSM and its implementation is needed by other work items. Also the implementation process and communication patters can be generalized across other work items observed. We make this generalization because all the observed

work items were implemented by developers of the same team using the same development tooling, settings and communication media; the only noticeable difference was the volume of communication. The methodology used to plot the communication evolution is based on the Social Network Analysis methodology. The generated diagrams are based on the information compiled from project planning documents, Bugzilla, and log of activities.

We observed the evolution of communication patterns during three specific stages of the implementation of the *Generic Folders* component. Our aim was to find gaps in the communication or other awareness-related problems that could lead to the generation of broken builds. The first stage observed was the planning of activities, the second was the interaction during design, and the third stage was the plotting of communication when the red build occurred.

1. Interaction of developers according to project planning

The assignment of work items in the planning documents is shown in the Figure 4.2. In the graphic the work items are represented by bug numbers. Developers explicitly assigned to work on a work item (bug) are within a square that represents the bug domain. Interaction between developers in the same domain is represented by arrows between them.

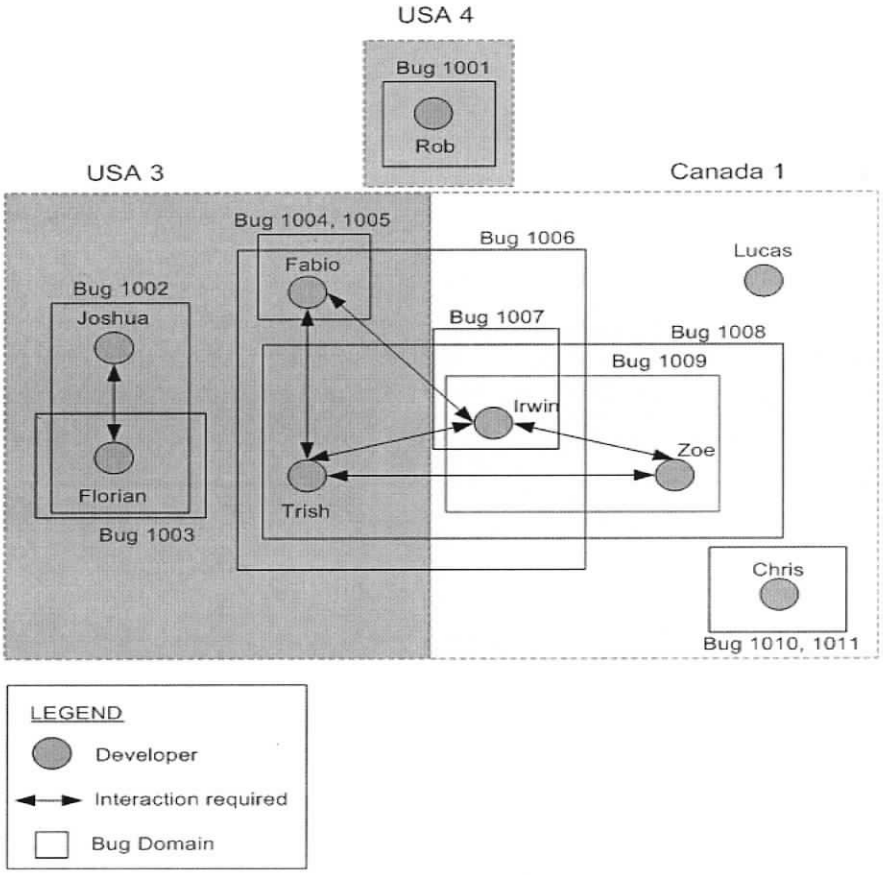

Figure 4.2: Work items (bugs) per developer included in the CSM plan as posted in the project WIKI

The CSM planning documentation explicitly mentions a collaborative relationship between Zoe and Irwin, in which Zoe was the main contributor to the Generic Folder work item (bug 1009). Typically, the responsible developer creates a new posting in Bugzilla to alert others that an implementation design proposal is ready for revision.

2. Interaction of developers during feature design

Figure 4.3 shows people that were part of the discussion about the work item implementation design. The dots represent people that posted a comment in Bugzilla (posters) and diamonds represent people that were alerted of postings (receivers). From the day that the bug was created until it was closed (14 business days), a total of 19 postings were submitted to Bugzilla.

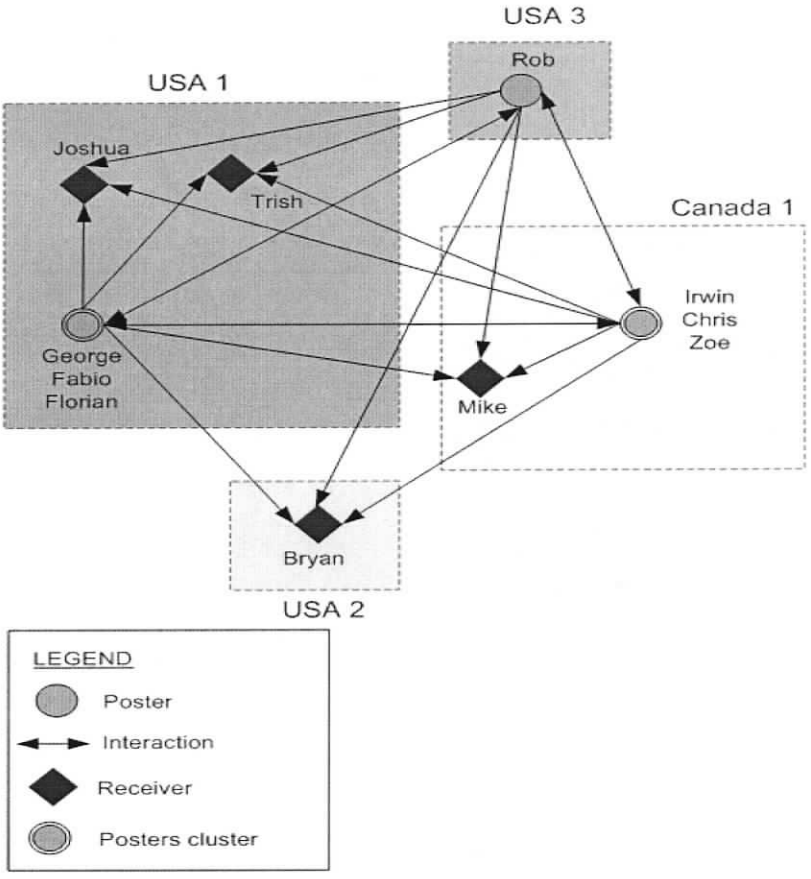

Figure 4.3: Communication exchange over Bugzilla for the bug 1009

Comparing Figure 4.2 and Figure 4.3, one can observe that the number of people involved in the discussion / implementation of the "Bug 1009" grew from 2 to 11. Typically, after recommendations are considered and the design improved, the implementation process begins.

3. Interaction of developers during implementation

The total number of people involved in the discussion / implementation / coordination of the work item was 17 as shown in Figure 4.4.

The graphic depicts a social network with information on communication media that supported the communication held among developers. The plot was generated from the communication logs provided by developers. We extracted the communication events related to the implementation of the work item 1009. Each arrow represents the communication between two developers and the edges represent the direction of the communication. Also, arrows contain information about the media and the date of the interaction. For example:

- F2F 11/28 means "Face to Face interaction occurred on November 28th"
- M 11/25 means "Email sent on November 25th"
- ST 11/21 means "SameTime chat on November 21st"
- P / ST 11/08 means "Phone call and SameTime chat on November 8th"

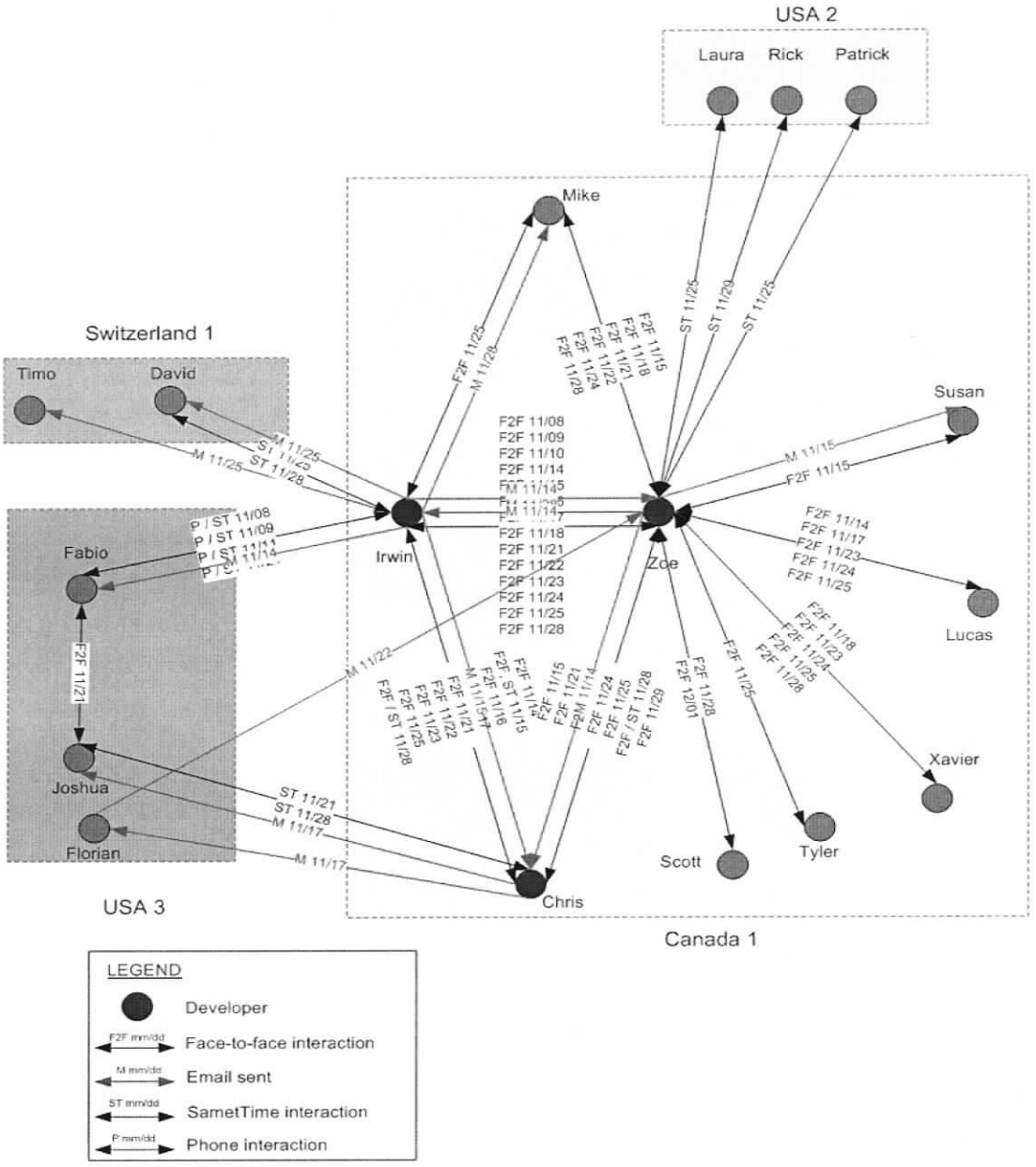

Figure 4.4: Social Network Analysis graphic based on the volume of communication generated during the Generic Folder implementation

We can also observe that the main developer (Zoe) had to deal with a high amount of information. The project relied on her personal initiative to implement and keep alerted concerning changes made using several different communication media. For example, Table 4.2 presents the amount of communication traffic related to the discussion / coordination and implementation of the work item 1009.

Table 4.2: Number of interactions by communication media for work item 1009

Communication media	Number of times used
Face-to-face	51
Bugzilla	19
Email	13
SameTime	14
Phone	4

4.2.2.3 The broken build and the volume of communication traffic

The creation of a more detailed study of the volume of information during the days of the generation of the broken build (November 14th) provided us with more insights into the volume of information that was handled by developers in the distributed teams. We found that in the days when the broken build was generated, the complete CSM team experienced a high volume of communication over the CSM mailing list, which can be translated into an intense development period (as it is shown in Figure 4.5).

This diagram depicts the number of messages sent by developers to the CSM mailing list from November 8th to December 1st. On November 14th the total number of

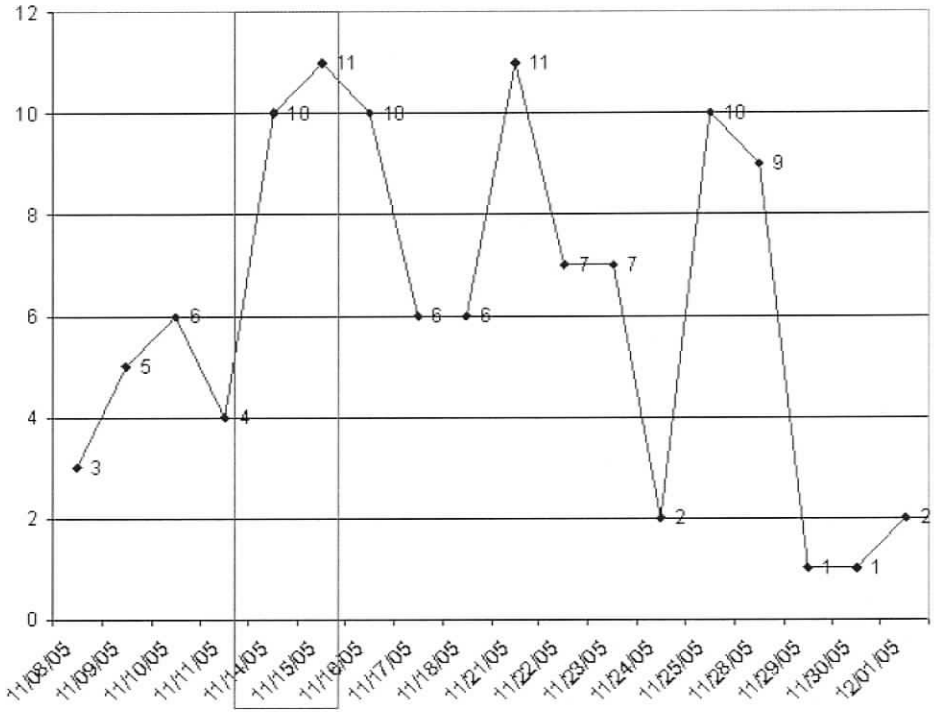

Figure 4.5: Number of messages sent to the CSM mailing list per day

messages sent to the list was 10, and on November 15th the total number of messages was 11. Regardless that the amount of communication does not seem overwhelming when you consider that this communication was only from one one of several work items being developed and that additionally, the developers were also receiving communication concerning the entire project, in fact, this amount of communication is significantly large.

4.2.2.4 What developers discussed about?

In order to categorize the information exchanged between team members shown in Figure 4.4, we classified the content of communication as reported by developers in communication logs. We inspected the communication logs to find keywords such as "sync-up," "coordinate," "change," "implementation," "planning," "risk," and "support," which helped us to classify the interactions. This classification was made by the author after finding patterns in the content of the communication between developers. The classification was verified by a colleague at the SEGAL Lab, however, no inter-coder reliability test was performed. The main categories identified were:

- **Implementation Issues**, for example, developers used SameTime to ask where they can find specific components in the repository.
- **Coordination of Activities**, for example, team leaders coordinated about which features needed to be implemented first by phone.
- **Communication of Changes**, for example, email sent announcing that the main data model had been updated and a new component had been added.
- **Synchronization**, for example, a quick face-to-face meetings between a developer and a team leader to communicate progress in an specific feature.

- **Planning**, for example, the team leader updated a planning document posted in the project WIKI.

- **Support**, for example, a developer asked to remote peer guidance in the creation of verticals in an specific component by email.

- **Risk Assessment**; for example, a developer sent an email to the mailing list asking if somebody was working in the implementation of features that could have potentially affected his workspace.

- **Other**; for example, developers ask about the status of projects that are not related to Team Tools at all.

After identifying the main categories, we extracted the volume of communication per category from the compiled activity logs. The results are shown in Table 4.3.

Table 4.3: Classification of issues and number of interactions by media for each category

Category	Media				**Number of interactions**	**Percentage**
	F2F	**Phone**	**ST**	**Email**		
Implementation issues	31	7	21	26	85	31.14
Coordination of activities	27	10	15	14	66	24.18
Communication of changes	9	2	8	26	45	16.48
Synchronization	22	5	8	1	36	13.19
Planning	13	3	1	3	20	7.33
Support	3	1	10	3	17	6.23
Risk Assessment	1	0	1	1	3	1.10
Other - not relevant	0	0	0	1	1	0.37

Table 4.3 shows that there are three significant categories, in which developers spend more time during their interactions: implementation issues, coordination of activities and communication of changes.

4.3 Summary

In this chapter we presented the findings of our research. The first finding shows the different communication media used by the developers of the Team Tools project. This finding shows that a high volume of communication is handled over face-to-face and phone, other communication media are important as well to support communication and collaboration. The findings about awareness are divided into two major sections; the first, describes awareness problems produced by the different corporate cultures and the second describes how handling a high volume of communication can create awareness problems and generate broken builds.

In the following chapter, we discuss the findings in the context of our research questions.

Chapter 5

Discussion

This chapter begins by discussing the findings of our case study in the context of our research questions in Sections 5.1 through 5.3. In section 5.4, we review the awareness problem according a new perspective after we completed our case study, and discuss if awareness problems can be tackled by improving software development practices. In Section 5.5, we present a discussion of the importance of awareness from the project management point of view, and how an awareness mechanism could help newcomers to come faster into speed. Our approach to tackle the awareness problem is presented in Section 5.6, and it includes the components of an awareness mechanism in an approach that is different. Finally, in Section 5.7, we present the threat to validity in our research as a way to express the limitations and shortcomings in our methodology.

5.1 Why communication volume matters in awareness?

In this section we revisit the first research question proposed in Section 3.1. In order to answer this question we will refer to the findings presented in Section 4.1., and also discuss the implications of our findings for the design of an awareness system. The research question:

> *Q1: How are developers alerted of changes during the development of a feature and which communication media are primarily used to propagate changes?*

Answering this question is very important in terms of the outcome of our research, considering that one of our aims is to identify information sources as an input of a feature-based awareness mechanism system. Our findings indicate different communication sources were used by developers in our case study. The developers were alerted about changes to features through notifications that were delivered using synchronous or asynchronous communication tools. The most relevant point of the findings is the high percentage of face-to-face and phone communication. We found that 34% of the communication occurs F2F and via phone.

This finding confirms other previous findings of empirical research that has shown that developers spend an average of 75 minutes each day in "unplanned interpersonal

interaction" [37]. It is however an important research finding: the fact that the information exchange over these media is neither recorded nor stored will impact the performance of an awareness mechanism system. The lack of the 34% of communication input, would not make the system 100% reliable. This finding triggers the challenge of defining business processes and tools to capture this information.

Considering the distributed nature of the project and the high volume of information over F2F and phone, we believe that relevant communication is not propagated to remote sites, leaving remote developers unaware of what is going on at the remote sites. Other communication media account for the remaining 66% of communication. These are: email, ST, mailing list, and Bugzilla. Our architectural design, which is presented in the next chapter, considers the addition of data repositories using APIs. Accessing the communication logs stored in repositories will imply the creation of algorithms to parse the structure of the messages to filter out if messages contain relevant information that is needed to be propagated among the contributors of the projects.

5.1.1 Bugzilla as an awareness tool

There is extensive research about the support of communication among distributed team members using asynchronous communication tools such as email [45], mailing list [40], and synchronous tools such as IRC [36]. However, the use of Bugzilla as a

communication tool to support development is particularly interesting in the observed project. This corroborates with evidence found in other studies [51][57] of distributed projects that Bugzilla is being used as a communication tool; hence, we believe that information sources such as Bugzilla need to be considered as an input of an awareness mechanism.

In the following section we will review our second research question.

5.2 The impact of culture differences on awareness propagation in a project

We would like to start this section with a sentence "being an IBMer does not really mean that I think as an IBMer." As we described in Section 3.2.1.4 the members of our case study came from different companies previously acquired by IBM. Both teams have their own corporate culture, language, and tooling set. Distance becomes an additional factor that team members cope with when they try to work out a solution to a conflict. Having this context, we will revisit our second research questions from Section 3.1:

Q2: Which awareness problems arise in distributed development settings?

In our case study, we observed that developers were not aware of changes on features made in remote workspaces. This lack of awareness was due to incorrect assumptions

about propagation of changes and the development practice of the remotely located team.

Perhaps the most challenging problem in distributed settings is coping with communication [18]; communication is the foundation of awareness. The findings in our case study illustrate how issues are raised and discussed over email, and how developers come up with possible solutions that can benefit themselves. The difficulties of communicating effectively across sites, led to a number of serious coordination problems. Among these problems were unrecognized conflicts among the assumption made at different sites and incorrect interpretation of communications, also found in the literature [37][18]. Research and empirical studies show that distance makes it more difficult to deal with organizational problems of a political and social nature [18]; and it is often difficult to integrate separate teams into a coherent team [9].

Observing the discussion on the mailing list provided us insights into communication problems caused by distance, and different corporate culture. We considered important to ask the team leader of the CSM component about his opinion on this discussion. He agreed that the implementation of this component is particularly challenging, not only on the technical site, but also because the Ottawa team is the lead-team chosen by management. On the other hand there is another team that according to its credentials would have been chosen to lead the development of the project. We realized that this discussion indicated the low level of trust between the

developers on both sites. The fact that the US team was not included in the project from the beginning harmed the morale on the US site; hence, the integration process had to overcome the extra challenge of dealing with a sensitive trusting relationship.

Despite all these challenges, team members were willing to work out possible solutions to overcome the barriers created by distance and corporate culture factors. This relationship building process is not specific to this project; this process actually takes place during all collaboration and communication between parties. However, relationship building practices are most important for projects with new partners and/or requiring quite constant collaboration [56]. In order to tackle similar problems, some companies have created strategies to foster the integration of remote teams and build trust based on literature. For example, this literature suggests, common kick-off meetings for the whole project when possible [10]. Another strategy is the creation of knowledge-sharing mechanisms. Having knowledge in a central repository can be structured in a way that teams, products, and divisions get a faster access to the required elements [21].

Without this kind of mechanisms, managers might fail to promptly and uniformly share information from customers and market among the development team [38]. Since IBM has been involved in large multi-distributed development projects, and probably some projects have experienced similar challenges, the experiences from previous projects are a valuable resource which managers can refer to when they face

similar situations.

In the following section we will review our third research question.

5.3 Collaboration patterns and awareness

Software development is an activity that requires extensive collaboration. Developers collaborate exchanging information through several communication media; hence, the collaboration relies on how well the developers communicate.

Collaboration patterns are based on communication patterns that arise within a development team. Understanding how communication tools are used and what information is propagated is very important when identifying design constraints of an awareness mechanism system that uses communication repositories as data input. In this Section we revisit our third research question from Section 3.1:

> *Q3: What are the collaboration patterns relevant in the study of awareness in feature management; i.e: roles and interdependencies between people and features?*

In our case study we observed that developers collaborate to coordinate activities, communicate changes, synchronize activities, plan actions, provide peer support and perform risk assessment (see Section 4.2.2.4). The information exchanged during these activities is vital to keep the team members aware of what is occurring in the

project. However, in terms of awareness, communication of changes is probably the most important collaboration pattern to consider.

Our findings show that the volume of communication exchanged to communicate changes is 16.48% of the total. We can observe that this percentage is quite significant, and a failure in the propagation of changes among the developers involved could generate problems that might lead to the generation of broken builds.

In Section 5.1, we discussed why the volume of information is important. Our findings suggest that the collaboration between developers generated a significant amount of communication that should be handled by developers per work item. It also heavily relies on personal skills to filter out relevant information, making this an error-prone and time consuming filtering process. As was explained in Section 4.2.2, one of the developers did not realize that an important message was sent to her; thus, inappropriate filtering of messages led to the generation of a broken build.

Having discussed the empirical evidence that directly addresses our research questions, we further consider that it is also necessary to review the awareness problem according to our understanding after this empirical investigation. In the following section, we will present a broader discussion about the awareness problem.

5.4 The problem of Awareness

In Chapter 2, we presented a review of literature on the awareness problem in software development. We covered implications of the definition of the problem of awareness, and different criteria to consider during the design and implementation of an awareness mechanism system. The theoretical background [61] and previous research [67] [7] [32] helped us understand the complexity of the problem. However, during our research we found that the awareness problem is more complex and challenging than we originally expected.

Before starting the interaction with developers in our case study, our understanding of awareness was limited to the use of artifacts and how changes to them can generate problems in the final deliverable of the system. After observing a development team during the system development process, we realized that problems due to the lack of awareness go beyond the changes to artifacts and their proper propagation. In fact, we identified two instances of awareness problems; they can either rise from non-well defined development processes and procedures, or from issues related to the use of development tools.

Both instances generate problems, not only with the system deliverables; but also within the team's structure, behavior, relationships, and morale. We will focus our discussion for the remainder of this section on finding answers to whether the awareness problem can be solved by improving development practices or providing

an automated mechanism to alert developers about changes to software artifacts.

5.4.1 Can awareness problems be tackled by improved software processes?

We believe that teams which have a well-defined set of processes and practices to support development and communication are better prepared to tackle awareness problems than other teams that do not have them. This argument is very obvious; however, is very difficult to implement because it takes a great amount of work and effort to do so. Considering that an in-depth discussion about process improvement to support awareness in software development goes beyond the scope of our research, we leave these thoughts as material that can be used for future research on how to tackle the awareness problem by improving software development processes and practices.

However, if we based a solution to the awareness problem on the implementation of a tool (or an extension of it), we must consider that a tool is created to support a business process. If the process is not properly designed, the tool will not warrant the solution of the awareness problem. Thus, before starting the design and implementation of a tool, the primary focus must be to understand the current process that will be supported to find the purpose of the tool.

The definition of new development processes is a managerial decision; hence, in the following section we present the management concerns about awareness.

5.5 Management concerns about awareness

Managers need to know what is going on in their projects. They must know what information is needed, where to locate it, and how to interpret and use it. Equally important is that they are able to do so without great effort [59]. If managers have to perform these actions, it is clear that they need an awareness system to assist them in compiling information to make decisions. It is even more important when all the resources and people are not physically located in same geographical area.

Project managers also cope with the Brooks' Law: "Adding manpower to a late software project makes it later" [4], and they try to find mechanisms to make newcomers come into the process faster without creating too much overhead for other team members, and delaying the project. Going forwards, we will consider as a newcomer either a team or an individual member. Sim [62] conducted a research in which project managers agree that in order to become productive, newcomers need to acquire a good and quick understanding of the system to work effectively.

In order to address management concerns within the awareness framework, we believe that in providing knowledge to newcomers about what is going on in the different workspaces that they have to interact with, they will find the necessary information in a more convenient, easier, and faster way. The result is obvious, newcomers will get up to speed faster avoiding overhead and project delays.

5.5.1 What should be given to the newcomers?

Information is the first answer to this question. However, information is a very broad and ambiguous term. Providing too little information will be perceived as a lack of collaboration. On the other hand, an overwhelming amount of information could be perceived as a delegation of activities that the remote teams do not want to do. The information provided to a newcomer should cover the questions that an awareness mechanism tries to answer [32]:

- Who is working on the project?
- What are they doing?
- Which artifacts do they manipulate?
- What is the impact of their work on others?

Besides information, a team should have an integration strategy of newcomers. This strategy should consider:

- Corporate culture of newcomer(s); we consider that the newcomers' culture must be understood by other team members,
- Information repositories used by newcomers, and
- Communication tools and protocols used within the newcomers' team and with remote teams.

Covering these aspects will give a team a baseline to improve their communication within the team and with remote teams. The benefits will be measured in the improvement of the development practice and the fewer broken builds originated by communication glitches or by being unaware of changes in other workspaces.

In the following section, we will present our tool-based approach to tackle the awareness problem based on our research findings, literature and experience.

5.6 Tackling the awareness problem

Our research has focused on understanding awareness problems created by changes in the development of particular features. Contributors, in the roles of designers, developers or testers, may work in seemingly separate workspaces, but their work is conceptually linked by a particular feature or set of interrelated features. We believe that the implementation of a feature-based awareness mechanism will provide timely access to current changes related to the contributor's work (either on demand or through a notification system). This mechanism, as a part of a configuration management system, will improve collaboration in software development to deliver notification of changes to features only to the contributors related to features. The use of the mechanism ultimately leads to better change management, productivity and software quality.

The proposed awareness mechanism has three well-defined components. The first

component, should compile change events that happen in information sources such as CVS, and Bugzilla. The second component should process the information compiled and create the conceptual links between contributors and features. The third one should deliver the information to the feature-linked contributors. In the following subsections we will cover each component in more detail.

5.6.1 Information Sources

We identified different available sources of feature change information in the software development environment, which led us to the following classification. This classification is similar to the classification of sources of awareness information in software development detailed by the work of Storey et al [68]:

5.6.1.1 Change Management tools

Depending on the size of the project, companies tend to set up configuration management tools. Medium and large sized projects tend to rely more on these tools to gain tighter control and track changes produced in the artifacts of the development process. These tools include: CVS repositories to control versioning of the code and documentation. These repositories also keep a record of changes and branching activities, allowing distribution of software revision and releases. Project repository logs contain information from a developer's local history (activities, artifacts used, etc.)

to provide awareness to other members about activities of other members of the same workspace or different workspace.

5.6.1.2 Defect tracking tools

Any software system is prone to have defects and bugs; these can be discovered during coding, testing and deploying development phases. These defects and/or bugs should be recorded in order to fix them or be kept as a valuable part of the documentation of the project. In a large project, it is more difficult to keep track of this information without using an automated system. Many companies design their own defect tracking system that allows the users to report a defect and/or bug and to report if a change was made in order to correct it. The most popular defect/bug tracking system is Bugzilla, an Open Source tool that is broadly used in many projects. Bugzilla has a mailing list as part of its mechanism; contributors who subscribed to it receive a notification when a defect/bug is reported. The disadvantage of this mechanism is that developer becomes overwhelmed by the vast amount of information that is simply not useful to them.

5.6.1.3 Documentation repositories

From the perspective of Requirements Engineering practice, requirements specification documents are considered to be the key to good software development practices

[65]. Detailed and timely requirements specifications act as a vital support to the development process, but if changes in the requirements are not properly documented or propagated, the system will be more likely to contain errors. For example, Requisite Pro has the functionality to store requirements in a requirement repository which can only be accessed for other Rational Tools, but this repository cannot be accessed by other development artifacts that require knowledge about the changes made in the requirements.

5.6.1.4 Communication repositories

- Asynchronous, as the main asynchronous communication media we could mention: mailing-lists, emails, discussion boards, Wikis.

- Synchronous, as the main synchronous communication media we could mention: audio conferencing, web conferencing, video conferencing, IRC, white boarding, application sharing.

Both communication types have advantages and disadvantages for awareness purposes. For example, by using asynchronous communication, contributors can always refer to a "physical" copy. By using an automated mechanism, the system will propagate changes to all parts related to these changes. With synchronous communication media, changes are broadcast in real time, so the propagation time is minimal. However, a major disadvantage is that not all the contributors affected by the changes

are notified in real time, nor are all of them even notified.

5.6.1.5 Know How

Electronic communication media and information stored in a project management repository are the primary information sources of an awareness mechanism system. However, "know-how" is often passed from more experienced members onto other developers by way of oral communication, for example, during face-to-face interactions. Also, project managers have information and "know-how" that could be very valuable for an awareness system. For example, a manager could provide information about management decisions that could affect the development of a project.

One of the challenges that we leave for future research is: How do we capture the "know-how" that is passed among contributors via informal face-to-face (or phone) interactions? At this point, we are fully aware of the richness of this data and the impact that it could have in the performance of our proposed awareness mechanism system.

5.6.2 Why is our approach different?

Awareness tools such as those reviewed in Section 2.3 and plug-ins for the Eclipse IDE [12] [46] have succeeded in providing awareness of code changes; however, the conceptual relationships between contributors are not made explicit in these tools,

nor are there notifications of changes to those involved in the development (and thus related) to a particular feature.

Other research projects [64] [58] [6] have succeeded in creating awareness mechanisms. However, the difference with our approach is that we aim to create an awareness mechanism that is based on what we refer to as conceptual relationships between features and contributors involved in the software development cycle. This is different from other research where conceptual relationships have been used to convey important dependency information to implementors and maintainers working with reusable software components [24] and to find document artifacts connected by hyperlinks based on the assumption that if two artifacts are linked to one another then there is some conceptual relationship between their contents [74].

An awareness system stores these relationships between the features and contributors in a relational database. Having this information stored in a repository, the mechanism will process and deliver different types of awareness (see Section 2.2.2) such as (Table 5.1).

Table 5.1: Types of awareness data

Type	**Description**
Change awareness	Knowing what changes were made in a feature that a contributor is working on and changes in other features that might cause an impact on his workspace.
Availability awareness	Identifying who performed changes in the features, and their availability for further clarification
Perspective awareness	Making impact analysis of feature changes in the contributor workspace, in order to coordinate collaborative work.
Environmental awareness	Focusing on events occurring outside the designers and/or developers' workspaces that might have implications for group activity.

5.6.3 Delivering the awareness data

Considering the different types of awareness data delivery as presented in Section 2.2.5, we believe that the delivery mechanisms should be *passive, differentiated, customizable*, and provide *peripheral* awareness data. One of the problems found in using only one notification system, such as email, is the result of an overload of information, also emails can get "lost" or "forgotten" [19]. For this reason, the notification of the awareness data should be distributed across multiple applications such as emails, pop-ups, SNA graphs, etc. The diversity of delivery applications will make the mechanism setting more flexible.

Graphic visualization is also important to deliver information about conceptual relationships between team members and artifacts. For this reason we consider that sociograms based on Social Network Analysis should be included in the delivery component of an awareness mechanism. Therefore, as part of our future research, we will create case studies, in which we can identify what delivery media are preferred by users of an awareness mechanism.

5.7 Threats to validity

5.7.1 Construct validity

5.7.1.1 Intentional validity

Do the constructs we chose capture what we intented to study?

To conceptualize problems as a result of lack of awareness, we chose broken builds as our research construct. The goal of our research was to get a better understanding of the awareness problem and to identify its causes. In our case study, we decided to observe a group of developers in order to describe the interaction process, and to find gaps in the development and communication that could generate broken builds. We believed that identifying these gaps and their effects in the final deliverables, we could come up with a solution at the tooling level to support developers and improve their productivity by reducing the time of communication of changes, and avoiding code rework. In addition to our personal research objectives, we observed the context of the project and the importance of having clean builds as a measure of success.

5.7.1.2 Representation validity

Do the constructs we chose translate well into observable phenomena?

This type of research was conducted for the first time at IBM; thus, the data collection process was permanently challenging. Specifically, the identification of

what information was needed to fulfill the defined construct. We monitored the status of the builds. The complexity of the generation of a build posed a challenge for the researcher; hence, we had to rely on the experience of developers and people in charge of the generation of the builds. Only they were able to find out the rationale of a failure in the generation of builds. Therefore, our understanding was unavoidably dependant of the explanation of the questioned developers.

5.7.2 Internal validity

Are the values of the dependant variable (red build) solely the result of the manipulations of the independant variable (level of awareness as generated by the volume of communication)?

During the data collection for our case study, we found evidence that an overwhelming amount of communication caused a developer to perform an action that generated a red build.

We believe there could be other alternate scenarios in which a red build could be generated, and they are described as follows:

The second possibility is the build generation setting; in Team Tools, the build generation procedure gathers objects from different CVS branches. We observed that it was a cause of continuous misunderstanding and frustration for developers because they did not know in which branch they had to commit the changes to their work. We

believe that developers can cause a red build due to a confusion of what and where they should commit their changes.

The third possibility that might cause a red build is failure in repositories; a bug can cause problems in Cruise Control, or CVS. For example, an error in versioning an object in the CVS can throw an exception, which will be captured immediately by Cruise Control stopping the build generation process immediately.

For the fourth possibility we have stated that developers rely on personal skills to filter important communication; however, in this assumption we considered that the developer is aware that information has been received and is waiting to be filtered. There is the possibility that developers might be distracted writing code and are not aware that there is a queue of information waiting to be filtered. In this case, the developer might generate a broken build by being unaware that information has already been posted.

A fifth possibility could be that after changes are done, the person responsible does not send any kind of notification. Thus, people related to the changes are totally unaware and perform actions that could lead to the generation of a broken build.

From the possibilities described in this section, we believe that the reason supported by our findings is the strongest cause of broken builds in the observed project.

5.7.3 External validity

Our research is unique in terms of having a researcher observing a real project under development. It is a rare opportunity that researchers are granted access to resources and information considered confidential or very sensitive. Under these particular circumstances, we dealt with issues such as figuring what to do with the information and how to gather only the necessary information without causing an overload to the developers involved. In solving this problem, our research faced some limitations in the definition of our case study and they are as follows:

5.7.3.1 Sample size and period of observation

The size of the project was quite large, it involved 60 developers distributed in North America and Europe, 12 research centres, several communication media, different tooling setting, multiple artifact repositories and thousands of lines of code. In our case study we dealt with a humongous amount of data, short timeframe, limited access to remote developers, and a lack of trust in the research. Despite these challenges, we received responses and support of a small group of developers (7 in total), who allowed us to observe their collaboration and communication patterns. Regardless of the size of our sample and the complexity of the observed component, we were able to identify some patterns that are presented in this document. However, we believe that if we could have observed a larger group of developers, we would have defined

more individual case studies and identified more patterns to make an evaluation and compare the results according the different scenarios observed.

The period of observation of the project limited the number of features and collaboration patterns observed. If we would have tracked a larger number of features during a longer period of time, we believe that we would have been able to find more collaboration patterns, and perhaps more broken builds caused by different awareness problems.

5.7.3.2 Research challenges on geographical multi-distributed projects

Multi-distributed geographical development projects cope with challenges related to distance and time zones. Conducting research in this type of projects faces the same problems that are inherited from the nature of geographical multi-distributed software development. Communication of research objectives was extremely hard with developers located in remote research centres. Most of the developers were not aware that our research was ongoing, so they were reluctant to participate and provide information about collaboration with their peers. As we observe, our research also coped with a problem generated by the lack of awareness of our objectives as researchers; developers made the incorrect assumption that if they disclosed information their jobs might be jeopardized.

Do the results of this study generalize beyond the organization studied?

The development practices observed at IBM is representative and quite general. We expect that other organizations would benefit from our findings; however, these organizations and development processes need similar characteristics to our case study, as they are described as follows:

First, the Team Tools project members are around 60, distributed in 12 locations across North America and Europe. However, we observed the development of only one component performed by a team of 7 highly skilled developers.

Second, the Team Tools project aims to develop a tool that will be used by developers, and these developers are in charge of designing, building and enhancing its functionality. This particular feature of the project allows developers to come up with ideas and initiatives which are implemented without having a formal design based on a requirement specifications document.

Third, Team Tools is implemented on top of the Eclipse IDE, and by some of the Eclipse development team members. Eclipse is an IBM Open Source Software (OSS) project. Team Tools has inherited the OSS management style of the Eclipse project. One of the development characteristics is the lack of formal requirements specification, as a foundation for the design and implementation of the tool. Therefore, we can say that Team Tools is a closed-source project managed as an OSS project.

Fourth, the distributed nature of the project and the high modularity of Team Tools make the project unique in terms of creation of collaborative patterns that

emerge according to the needs of the people involved in the development. These collaboration and communication patterns can be generalized in terms of tooling support. Most of the communication tools used in the project are generic: email, chat, mailing lists, phone, Bugzilla, and face to face. Although, we can not expect the type and volume of communication to be similar in other projects.

Considering that companies are embracing Agile development processes, and the impact of OSS projects is shaping the development style in closed-source projects[47], we can say that our results are likely very applicable to projects and organizations with similar characteristics, management style and multi-site distribution. However, practitioners and companies should be aware that although they may obtain similar findings, the solution is not necessarily the same; they will need to find a solution to the problems found by reviewing their own development process and practices.

5.8 Summary

In this chapter we have revisited our research questions, and the implications of the findings trying to answer them. The awareness problem was revisited from a post-research perspective. In this section we stated that the awareness problem should be tackled by improving development processes and practices supported by tools designed according to the needs of a particular project. Based on the findings of our research, we introduced our approach of an awareness mechanism should process rel-

evant information about changes in artifact repositories and create conceptual links between contributors and software artifacts. Finally, we closed this chapter presenting the limitations of our research expressed in terms of threats to validity in our research. In the following chapter, we will present the functional and non-functional requirements, and the architecture of a feature-based Awareness Mechanisms System based on our case study findings and our tool-based approach to tackle the awareness problem.

Chapter 6

Architecture of a feature-based awareness mechanism system

In this chapter we present the architecture of a feature-based Awareness Mechanism system as a tool-based approach to tackle the awareness problem described in previous chapters. From literature and findings of our case study we draw the functional and non-functional requirements, which are presented in Section 6.1. From the requirements we define our Use Case Model that is included in Section 6.2. The system architecture is introduced in Section 6.3. The architecture contains objects that will collect events in data repositories, process and create conceptual relationships and alert only contributors that belong to the same conceptual network.

6.1 Awareness Mechanism System Requirements

One of the aims of our research is to design a better awareness mechanism to support the notification of changes during software development. For this reason we conducted a case study to have a better understanding of the awareness problem, and to draw the requirements for such a system. In this section, based on our approach to tackle the awareness problem described in Section 5.7 and literature, we define the functional and non-functional requirements that will guide the design of a feature-based Awareness Mechanism System.

6.1.1 Functional Requirements

A feature-based Awareness Mechanism System must comply according to three major functional requirements:

6.1.1.1 Awareness data collection

The system shall collect events related to feature changes occurring in the artifacts and communication repositories.

Rationale: The system shall capture information related to a feature change occurring in artifact repositories such as: CVS, Bugzilla, communication repositories and others. The collected information must include the type of event, the feature identification, and the information of the contributor who performed the change.

6.1.1.2 Awareness data process and creation of conceptual relationships

The system shall process the collected information and create conceptual relationships to link contributors related to the implementation of a feature.

Rationale: The system shall process the information collected (types of events, features, and contributors) and create a network of contributors involved in the development of the feature provided by the collection component. The contributors should be linked through features. The links between features are defined on requirements and design documentation, and code ownership.

6.1.1.3 Awareness data delivery

The system shall deliver awareness data to the contributors that belong to the same conceptual network according to notification rules defined by the contributor.

Rationale: The system shall propagate awareness data to the contributors that belong to the same feature-based conceptual network. The notification must be according to rules pre-defined by the contributor. If the system does not find a contributor pre-defined notification rule, it will choose a generic system notification rule.

6.1.2 Non-Functional Requirements

There are some Non-Functional requirements that should be taken in consideration in the implementation of an awareness mechanism system. These requirements are

based on the characteristics of an awareness system introduced by the Palantir project [58] as reviewed in Section 2.2.4. An awareness mechanism can gather information "explicitly," by asking the user to provide it, or "implicitly" if the awareness system collects information from the system [67]. Our approach relies on the following principles for supporting workspace awareness [22]:

1. Awareness information should be present in the same-shared workspace as the object of collaboration. Awareness information must be easily interpretable regardless of where it is presented.

2. Awareness information should be passively collected and distributed by the system, rather than explicitly provided by the participants.

The following AMS non-functional requirements are extracted from literature and previous research [58] [67]. They are:

6.1.2.1 The awareness mechanism should be non-obtrusive

The architecture of the awareness mechanism should consider that the activities performed by the user will not be affected by using the AMS. The feature-based Awareness Mechanism System should fit an existing development and communication environment without replacing a current functionality, interfering with existing processes, nor deprecating their performance.

6.1.2.2 The awareness mechanism should be scalable

The performance of the AMS should be totally independent of the volume of data that the system retrieves, processes and delivers. Also the impact of the AMS on existing systems should be kept to a minimum.

6.1.2.3 The awareness mechanism should allow configuring the delivery of the information

The user of the AMS should be able to filter the information and define when it will be presented without requiring too much manual configuration on the user part and sacrificing usability.

6.2 Use Case Model of an Awareness Mechanism System

The AMS system will collect a set of events produced in the data repository; the collection of the events will occur through monitoring of the actions performed by the contributors to the project. When an event is relevant for a contributor the, AMS will create AwarenessNotification objects to alert about the events collected by the AwarenessEvent object. Figure 6.1 displays the sequence of processes since the collection of events until the notification delivery.

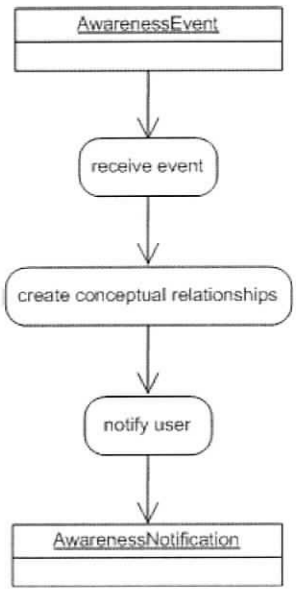

Figure 6.1: Process from the event capture until its delivery

The main processes of an AMS can be described in terms of three use cases (see Figure 6.2): collection of events, process and creation of feature-based conceptual relationships and delivery of the notification. The main actors of an awareness system are the Subject (user that originated the event), and the Receiver (user that is alerted about an event).

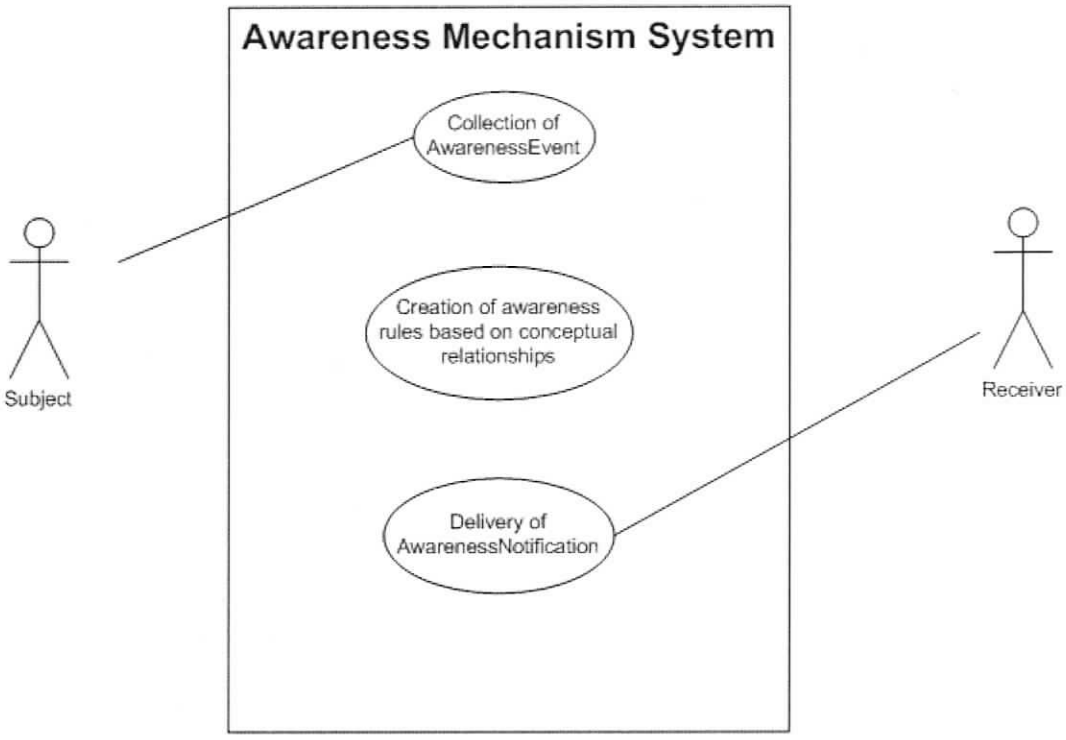

Figure 6.2: Use Cases diagram of an AMS

The three cases can become the three services as shown in Figure 6.3. This will be part a of the feature-based Awareness Mechanism System.

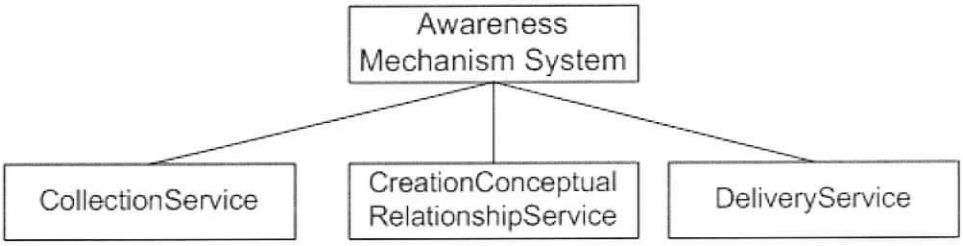

Figure 6.3: Services of an AMS

6.3 Architecture of the Awareness Mechanism System

The AMS can be split into three subsystems based on the three services defined in the previous section. Each subsystem is in charge of performing a set of actions that will be covered later in this section. The subsystems are shown in Figure 6.4.

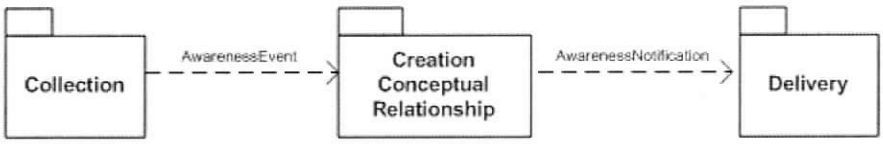

Figure 6.4: AMS Subsystems

Each subsystem receives input, performs actions and sends a set of output to the next subsystem. The Collection subsystem will retrieve information from the information source defined for the system (i.e.: Bugzilla, and CVS), the Filtering

subsystem parses the information according to a set of rules stored in the server. These rules are previously defined for the user of the system, and the Delivery subsystem generates the final display of the data according to a pre-defined set if rules defined for the event that happened in the repositories.

6.3.1 Distribution of the AMS components

The AMS system has a set of components that will run in the AMS server and other individual components that will run in the client sides. Figure 6.5 shows the distribution of components between the AMS server and clients.

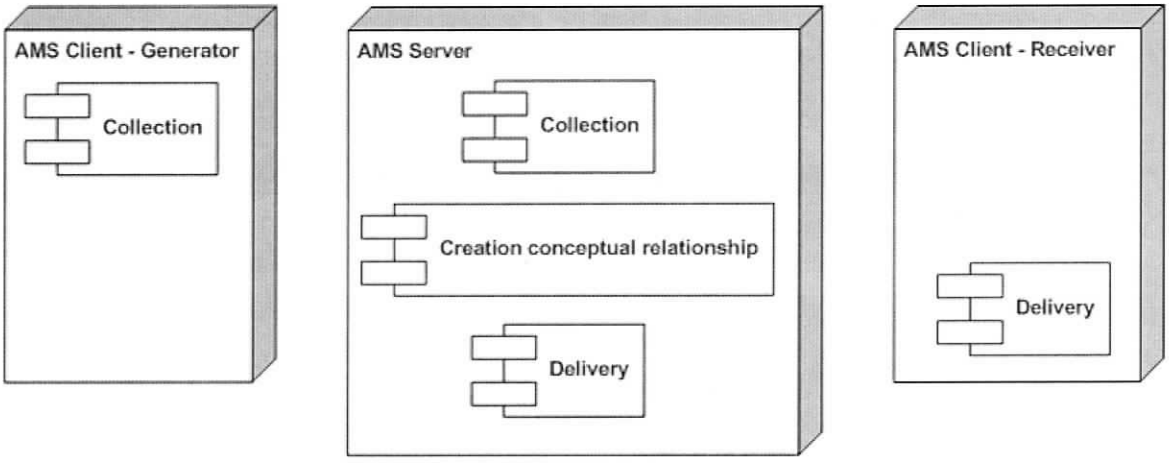

Figure 6.5: Distribution of the components in the AMS server and clients

6.3.2 The AMS Server

The AMS server implements the core functionality of our system. Figure 6.6 outlines the staged event processing. When an event is captured in the client Generator (i.e.: a commit in the CVS repository), the information related to the event is processed and conceptual relationships between contributors and features are created. After the network of contributors is created, notification rules are retrieved from a repository that contains user-defined rules. If the rule matches the parameters that originated the event, the notification process creates a notificationEvent, which is put in queue for delivery. The exception to this flow, is the non-existence of a user defined rule. In this case the system will apply generic rules stored in a system defined rules repository. Finally, the system delivers the notifications only to the contributors that belong to the same feature-based conceptual network generated. The delivery information is taken by the client Receiver, which creates a meaningful display or alert for the contributor.

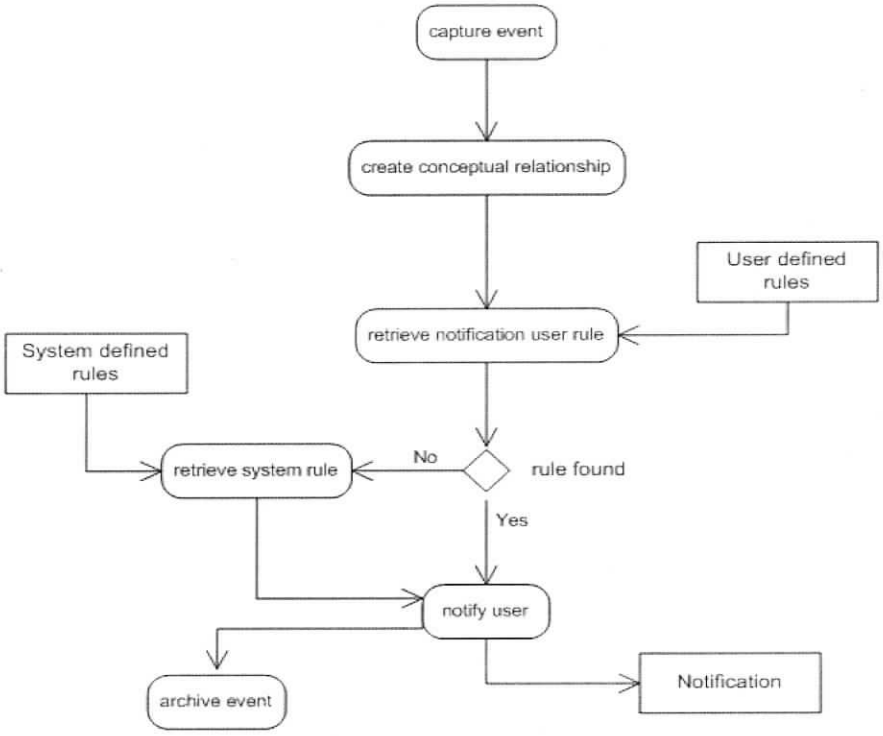

Figure 6.6: An outline of activities involved in the processing of events

The process is supported by data that is stored in a relational database back-end. Figure 6.7 shows the architecture of the database schema that will contain this information. The database back-end contains tables that will store the necessary information to create the conceptual relationships between contributors and features. We have defined tables CONTRIBUTORS and FEATURES to store individual information of both. The table FEATURE-CONTRIBUTOR stores the relationship between contributors and features. This relationship is defined by the information extracted from requirements and design documentation, and code ownership.

There is not any direct link between contributors, as we observed in our case study. Contributors are assigned to features and features are linked to other features (for example, in our case study, we observed that the CSM component was implemented by 12 features). Thus, contributors are linked through features. To create the feature-to-feature link in our architecture, we have created an internal reference key in the table FEATURES, and an extra optional field called FEATURE-ID-CHILD-OF. This field will store the reference of a feature to another one.

The other tables included in the architecture will store the information about the notification rules defined by the contributor. At this point in our research, we have not defined these rules. They will need to be defined in future research; however, we believe that having these notification rules as part of the database will improve the customization of the awareness data delivery.

We have also included the table AWARENESS-EVENT-LOG to store all the notification events that occurred in the system to keep abreast of the evolution of the system. This will help to generate statistics about the system performance and will facilitate audit tasks.

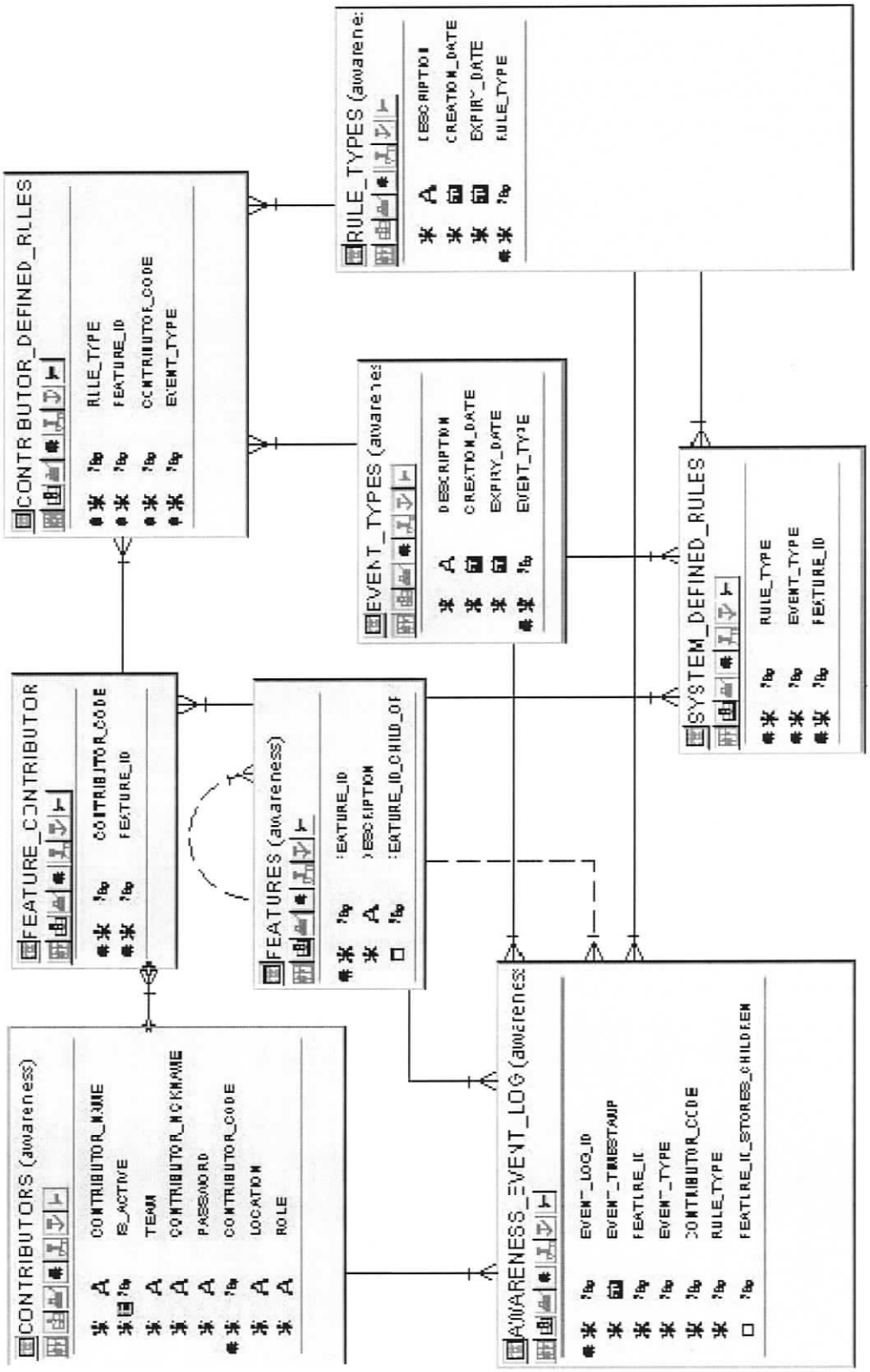

Figure 6.7: AMS Server database architecture

6.3.2.1 AMS server tables description

- CONTRIBUTORS: information of contributors of a project.
- FEATURES: information concerning a feature.
- FEATURE CONTRIBUTOR: the conceptual relationship between contributors and features.
- EVENT TYPES: types of possible events that can be triggered by an action in data repositories, (i.e.: a commit in the CVS repository, or a posting in Bugzilla. This set of events can be defined by observing and analyzing a development process. Each important event can be categorized and a code can be generated for each).
- RULE TYPE: types of rules related to the notification of an event (i.e: send a notification via email immediately after an event occurred). The information that will populate this table should be defined in a case study, where contributors have to define the notification rules.
- CONTRIBUTOR DEFINED RULES: a unique combination of the awareness rule defined by a user when an event occurred to a specific feature.
- SYSTEM DEFINED RULES: a unique combination of rules applied when an event happens to a specific feature.

- AWARENESS EVENT LOG: historical information about past events.

6.3.2.2 The AMS Logical Model

The logical model (see Figure 6.8) has been designed as a generic approach to collect events from different repositories. For this specific phase of the design we have considered objects that capture events in Bugzilla and CVS repositories. The CollectEventCVS, and CollectEventBugzilla objects collect events happening in the CVS and Bugzilla repositories respectively. Each event is linked to a particular feature, and it is produced by a contributor. The event, feature, and contributor are passed onto the EventProcess object, which using the IRepositoryDataMediator interface will retrieve information from the database in order to create the feature-based conceptual network. The same object will retrieve the awareness rule that matches the combination "Contributor + Feature + Event" from the database. After the EventProcess object has finished the creation of the conceptual network, it will pass the notification action onto the EventDelivery object, which handles the notifications queue. The INotificationHandler interface directs the awareness notification to the respective NotificationHandler objects such as: EmailNotifiactionHandler or PopupNotificationHandler.

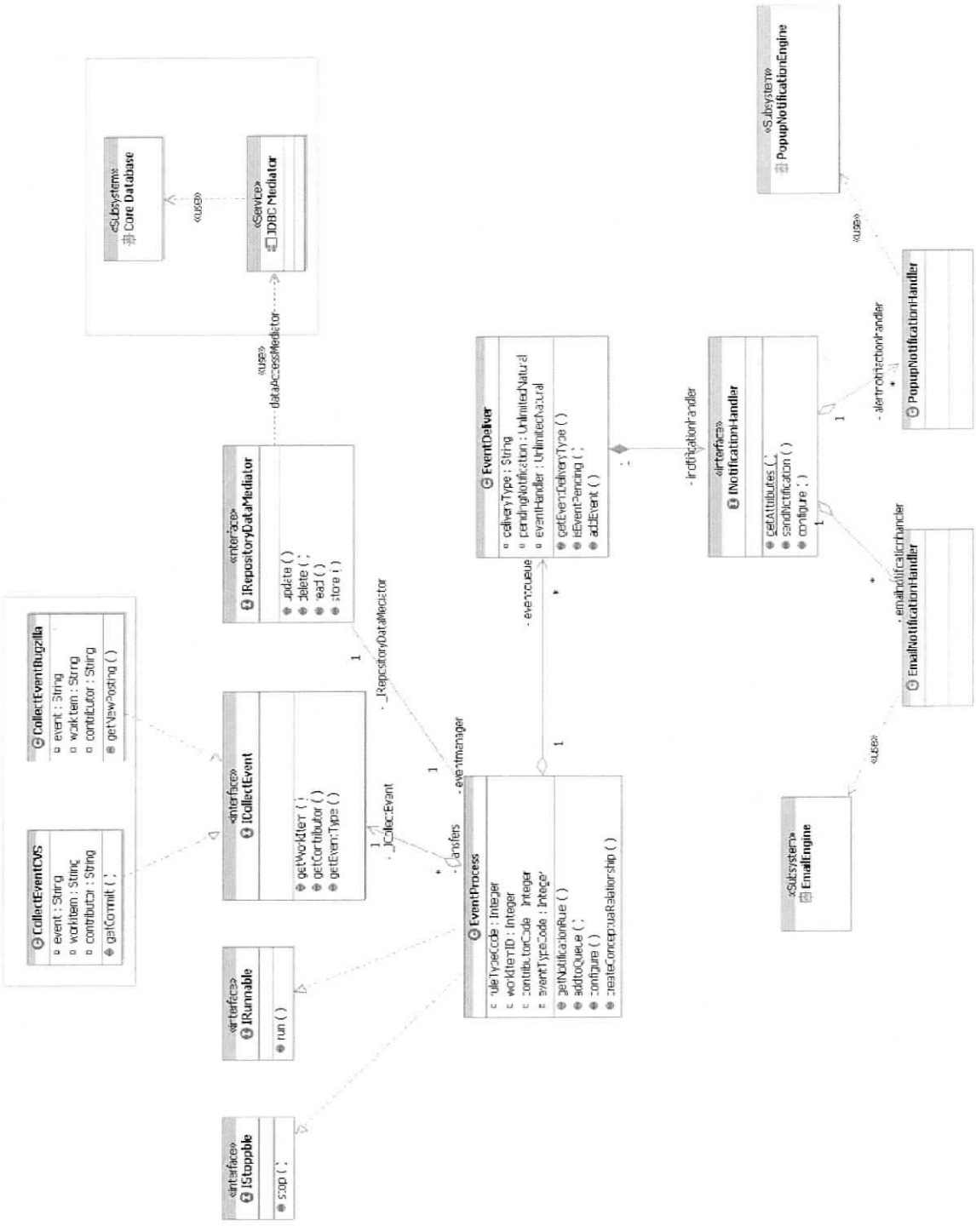

Figure 6.8: AMS overall architecture

We have designed the architecture of a feature-based Awareness Mechanism System. As noted in Figure 6.8, we can continue extending the number of objects in order to collect events from other repositories such as communication logs, or CASE design tools. Another advantage of this architecture is that AMS objects can be added as part of current distributed team support tools such as IBM's Team Tools, or Sysiphus [73].

6.4 Summary

In this chapter we have proposed the architecture of a feature-based Awareness Mechanism System. We started this chapter presenting the functional and non-functional requirements of the AMS, and based on them, we created the use case which contains three main services: the collection, the creation of conceptual relationships, and the delivery. The AMS will distribute the components between the client and the server. The AMS server will have a relational database as a back-end to store the relationships between features and contributors and also the notification rules. In the following chapter we present the conclusions of our research.

Chapter 7

Conclusions

In this chapter we present the conclusions of our research and experience. In Section 7.1, we provide insights to understand the awareness problem in the context of our case study. In Section 7.2, we review our research in relation to our research questions. In Section 7.3, we present the contribution of our research. Directions for future research are provided in Section 7.4. Finally, we conclude this chapter with concluding remarks.

7.1 Insights

Our research presents the awareness problem from the theoretical point of view, as well as the practical perspective, based on an empirical observation of software development practices in one of the IBM Research Centres in Canada. In this section we

review more insights of the awareness problem and its complexity. We provide final remarks on the importance of communication in the awareness context, and the implications of having different corporate cultures in the propagation of changes during development.

7.1.1 The awareness problem and organization structure

The awareness problem is very complex, and it must be tackled through improved software development processes and practices using tool support. Besides processes and practices, another important aspect to consider is the organizational structure in which changes need to be propagated. Having a good understanding of the organizational structure will help us to model an AMS architecture that considers different organization layers (i.e.: management, development, etc.), and contributor roles that belong to each layer.

In our case study, we identified only two well-defined roles: developers and team leaders; and three organization layers: development, team leading, and project management. Thus, the complexity of the propagation of changes was limited to two roles and three organization layers. However, this model could grow in dimension and complexity depending of the number of contributors involved and the nature of the system that is built.

Finally, we believe that if changes are not properly propagated within the or-

ganizational structure, then they might cause project delays and possible failures. Measuring the impact of the problem in economic terms, we believe that an investment on the implementation of improved processes and tools such as AMS to support them, will bring a positive revenue to the organization that assumes the endeavor of deploying this system as a part of its software development practice.

7.1.2 Awareness and communication

Communication is the foundation of awareness. Furthermore, communication that is not propagated in a timely manner to the concerned contributors is useless in terms of alerting about changes to features during the development process. During our case study, we observed communication patterns in the use of communication tools to support development. According to our findings, we believe that the fact that a significant percentage of the communication is via phone and face-to-face, poses a challenge for any awareness mechanism that uses communication sources to create conceptual relationships among contributors (because this information is neither recorded nor stored). Consequently, the information delivered by an awareness mechanism would not be 100% accurate and realiable.

The use of Bugzilla as a communication media makes this tool another important source of information for an awareness mechanism. As we referred in our discussion in Section 3.4.4, we did not find that Bugzilla was used as a tool to host discussions

around the design of features.

7.1.3 Awareness and corporate culture

Corporate cultures are different among teams, even if they belong to the same company or corporation. These differences become more noticeable when the teams are geographically distributed. Alerting of changes in shared repositories by multi-distributed teams becomes a challenge because the propagation of information must consider important aspects related to the culture of the teams involved in the development process. These aspects are: communication style, development styles, communication and development tools. We observed that in distributed development, the lack of knowledge of the culture of another remotely located team leads to the generation of wrong assumptions during collaboration. Thus, the classic question that is intended to be solved by an awareness mechanisms rises "what is going on?" remains unsolved and generates problems in the software implementation and in creating a homogeneous culture across the organization.

We believe that the implementation of a feature-based Awareness Mechanism System will support distributed teams by providing awareness of personalized information about changes to features made by users of the system regardless of their location. Developers alerted about changes to features can track down the changes and link them to the artifacts and the contributors that modified them. Thus, a

network of people linked through features can be created beyond geographical limitations. These networks will provide to contributors more information about what is happening in remote sites; providing remote awareness will foster the creation of a common corporate culture.

7.2 Reviewing the Research

Having reviewed the insights of our findings, we review how our research questions have been addressed.

7.2.1 How are developers alerted of changes during the development of a feature and which communication media are primarily used to propagate changes?

According to our findings, developers rely on several communication media to propagate changes. We found that communication of changes is mostly propagated via email, either by messages sent to mailing lists or messages sent to specific developers. Moreover, the importance of this finding is the identification of communication sources as input of an AMS. Considering this purpose, and observing again the settings of our case study, the most important result is that the 34% of the information handled by the team was over phone and face-to-face interactions. Hence, we can con-

clude that an AMS will lack a high percentage of communication input. This finding triggers other research questions to find solutions to how to capture this information that could contain important and valuable content for the success of the project. We also observed that variety of communication tools used during the development process, in fact the 66% of the communication, is over other tools such as: email, mailing lists, SameTime, and Bugzilla. This volume of communication can be leveraged for the AMS as data input.

7.2.2 Which awareness problems arise in distributed development settings?

The distributed nature of our case study provided us a framework to observe awareness problems produced by geographical distance between team members. The main awareness problem found was the lack of knowledge of "what is going on in the remote site?". Not knowing the status of the activities of others, and in particular to changes that affect a contributors' workspace in remote sites led, to false assumptions which, in turn, led to incorrect decisions. The problem of distance and speed of propagation of changes is deeply linked to the lack of knowledge of corporate cultures of remote teams.

7.2.3 What are the collaboration patterns relevant in the study of awareness in feature management, i.e: roles and interdependencies between people and features?

We stated that the collaboration patterns are based on the communication patterns observed within the case study's development team. We then observed that a high volume of communication of changes is handled for each developer; thus, all the responsibility to filter relevant information relies on the personal skills of each developer. We also observed that developers collaborate on the implementation of a work item. This collaboration creates a network of people based on the communication and artifacts. The network provides us some insights about collaboration patterns among developers and helps us to identify roles within the network members such as: main developers, active contributors, receivers, and clients of the work item.

7.3 Contributions

The contributions of this thesis are important toward building a better understanding of the awareness problem and possible solutions to tackle it. The most significant contributions are:

- The awareness problem has been largely discussed in the literature, however our thesis presents insights from a closed-source industrial software develop-

ment project. We have documented the evidence of our findings, which shows that awareness problems are the result of several factors related to geographical distribution, collaboration patterns, corporate culture, and communication overload.

The description of the research setup is also an important component of this contributions. Investigating day-to-day collaboration practice of a software development team is not a trivial task. Our case study described the instruments devised to collect information about types of communication that are practically almost impossible to capture, e.g. telephone and face-to-face communications. While imperfect our method may guide researchers in conducting similar studies and improving the methods of collection of such data.

- We have presented an architectural design of an Awareness Mechanism System that aims to provide awareness to contributors involved in the development of a project by creating what we referred as conceptual relationships between features and the contributors that modified them. An AMS will collect information from identified sources, create conceptual relationships, and propagate awareness information to all the contributors involved in the implementation of a feature and which are conceptually related by the individual work performed in software artifacts linked to a feature.

7.4 Directions for Future Research

Our research is the beginning of a large and ambitious endeavor that we expect will be followed by practitioners interested on tackling the awareness problem. Throughout this document we have mentioned that the awareness problem challenges us to design better development processes and tools to support them. In this section, we present some topics that deserve further research in order to come up with more solutions and more ideas about how to cope with the awareness problem.

1. We need to design a case study to identify the structure of the information that should be extracted from the multiple information sources identified in Section 5.7.1. Then, mechanisms and algorithms need to be designed in order to extract the information and make it available to the AMS.

2. The conceptual networks generated by the AMS must be validated with developers in terms of accuracy. We need to validate if the network is capturing the contributors involved in the implementation of a particular feature.

3. We need to design a case study to identify appropriate awareness delivery mechanisms that contributors need without causing additional overhead.

4. After the AMS has been deployed, we need to direct a case study that compares whether teams using the AMS do handle changes better than teams that do not use it and improve project member's productivity. We will measure productivity

improvement in terms of less time of code rework, and less number of broken builds.

7.5 Closing Remarks

At the beginning of our research we had limited understanding of the complexity of the awareness problem and how it affects the software development practice within an organization. During our research, we found that the complexity is related to several factors such as processes, collaboration, communication, development, and information sources. In our findings we have presented how awareness problems arise when people collaborate. We also found that people are involved in the solution of awareness problems, which means that the human factor is an extremely important component to consider when we design and implement an integrated solution that will tackle the awareness problem from the business process and tooling perspective. We believe that more research has to be done in order to corroborate our findings and enrich the awareness theories to provide more insights from industrial projects.

Bibliography

[1] L. Adamic and E. Adar. How to search a social network. *Technical Report HP Labs, Palo Alto*, 2003.

[2] J.A. Barnes. Class and committees in a norwegian island parish. *Human Relations, Vol. VII*, pages 39–58, 1955.

[3] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 1976.

[4] F. Brooks. *The Mythical Man Month.* Reading, Mass.:Addison-Wesley, 1975.

[5] F. Brooks. No silver bullet: essence and accidents of software engineering. *Computer, v.20 n.4*, pages 10–19, 1987.

[6] J.J. Cadiz, S.R. Fussell, R.E. Kraut, F.J. Lerch, and W.L. Scherlis. The awareness monitor: A coordination tool for asynchronous, distributed work teams. *Available from www.cs.cmu.edu/kraut*, 2000.

[7] J.J. Cadiz, G. Venolia, G. Jancke, and A. Gupta. Designing and deploying an information awareness interface. In *Proceedings of CSCW*, pages 314–323, 2002.

[8] J.A. Cannon-Bowers, E. Salas, and S.A. Converse. Shared mental models in expert decision-making teams. *In N.J. Castellan, Jr. (Ed.), Current issues in individual and group decision making*, pages 221–246, 1993.

[9] E. Carmel. *Global Software Teams: Collaboration Across Borders, and Time Zones.* Prentince-Hall, Uppers Saddle River, N.J., 1999.

[10] E. Carmel and R. Agarwal. Tactical approaches for alleviating distance in global software development. *IEEE Software*, pages 22–29, March/April 2001.

[11] L. Catledge and C. Potts. Collaboration during conceptual design. In *Proceedings of the 2nd Int. Conf. Requirements Engineering., ICRE'96*, pages 182–189, 1996.

[12] L. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. *In Proceedings of 2003 OOPSLA Workshop on Eclipse Technology eXchange. ACM*, pages 45–49, 2003.

[13] H. Cho, M. Stefanone, and G. Gay. Social network analysis of information sharing networks in a cscl community. In *Proceedings of Computer Support for Collaborative Learning (CSCL) Conference*, pages 43–50. Mahwah, NJ: Lawrence Erlbaum, January 2002.

[14] A.V. Cicourel. The integration of distributed knowledge in collaborative medical diagnosis. *In J. Galegher, R. E. Kraut, & C. Egido (Eds.), Intellectual Teamwork: Social and Technical Foundations of Cooperative WorkHillsdale, NJ: Lawrence Erlbaum Associates*, pages 214–242, 1990.

[15] R. Cross, S. Borgatti, and A. Parker. Making invisible work visible: Using social network analysis to support strategic collaboration. *California Management Review*, pages 25–46, Winter 2002.

[16] R. Cross, N. Nohria, and A. Parker. Six myths about informal networks and how to overcome them. *MIT Sloan Management Review*, 43(3):67–76, Spring 2002.

[17] D. Damian, J. Chisan, P. Allen, and B. Corrie. Awareness meets requirements management: Awareness needs in global software development. *Global Software Development Workshop, International Conference on Software Engineering, Portland, Oregon*, 2003.

[18] D. Damian and D. Zowghi. An insight into the interplay between culture, conflict and distance in globally distributed requirements negotiations. *Proceedings of the 36th Hawaii Conference on Systems Sciences (HICSS'36)*, January 2003.

[19] D. Damian and D. Zowghi. Requirements engineering challenges in multi-site software development organizations. *Requirements Engineering Journal*, 8(3):149–160, 2003.

[20] Allan Davis. *Software Requirements: Analysis and Specification.* Prentice-Hall, 1990.

[21] K.C. DeSouza and J.R. Evaristo. Managing knowledge in distributed projects. *Communications of the ACM*, 47(4):87–91, 2004.

[22] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of WEbNet'96*, pages 107–114, October 1996.

[23] P. Dourish and S. Bly. Portholes: Supporting awareness in a distributed work group. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1992)*, pages 541–547, 1992.

[24] S.H. Edwards, D.S. Gibson, B.W. Weide, and S. Zhupanov. Software component relationships. In *Proceedings of the 8th Annual Workshop on Software Reuse*, 1997.

[25] M. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64, 1995.

[26] A. Espinosa, J. Cadiz, L. Rico-Gutierres, R. Kraut, W. Scherlis, and G. Lautenbacher. Coming to the wrong decision quickly: Why the awareness tools must be matched with appropiate tasks. *CHI Letters, volume 2, issue 1*, pages 1–6, 2000.

[27] J.M. Gonzalez-Barahona, L. Lopez, and G. Robles. Community structure of modules in the apache project. *In Proceedings of the 4th International Workshop on Open Source Software Engineering. Edinburgh Scotland*, pages 44–48, 2004.

[28] The Standish Group. The chaos report. *The Standish Group International*, 1994.

[29] The Standish Group. Extreme chaos. *The Standish Group International*, 2001.

[30] C. Gutwin and S. Greenberg. Workspace awareness for groupware. In *Proceedings of the CHI'96 Conference on Human Factors in Computing Systems*, pages 206–207, 1996.

[31] C. Gutwin and S. Greenberg. A descriptive framework of workspace awareness for real-time groupware. *JCSCW, Issue 3-4*, pages 411–446, 2002.

[32] C. Gutwin, R. Penner, and K. Schneider. Knowledge sharing in software engineering: Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, November 2004.

[33] C. Gutwin, M. Roseman, and S. Greenberg. A usablity study of awareness widgets in a shared workspace groupwork system. In *Proceedings of the 1996 ACM conference on supported cooperatice work*, pages 258–267, November 1996.

[34] F. Harary. *Graph Theory.* Reading, MA: Addison-Wesley, 1969.

[35] J.D. Herbsleb and R. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference of Software Engineering*, pages 85–95, 1999.

[36] J. Hersleb, D. Atkins, D. Boyer, M. Handel, and T. Finholt. Introducing instant messaging and chat in the workplace. *In CHI'02*, pages 171–178, 2002.

[37] J. Hersleb, T. Finholt, and R. Grinter. An empirical study of global software development: Distance and speed. *In International Conference on Software Engineering*, pages 81–90, May 2001.

[38] J. Hersleb and D. Moitra. Global software development. *Editorial issue on GSD, IEEE Software*, 18(2):16–20, May 2001.

[39] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. *In Proceedings of the ACM 2004 Conference on Computer Supported Cooperative Work*, pages 444–454, 2004.

[40] L. Izquierdo, D. Damian, and D.M. German. Towards understanding requirements management in a special case of global software development: A study of asynchronous communication in the open source community. In *Proceedings of the International Workshop on Distributed Software Development*, pages 171–185, August 2005.

[41] Michael Jackson. *Software Requirements & Specifications: Practice, Principles, and Prejudices.* Addison-Wesley Pub. Co., Boston, MA, 1995.

[42] R. Kobylinski, O. Creighton, A.H. Dutoit, and B. Bruegge. Building awareness in global software engineering projects: Using issues as context. *International Workshop on Global Software Development (co-located with ICSE '02)*, 2002.

[43] Gerald Kontoya and Ian Somerville. *Software Requirements: Processes and Techniques.* John Wiley and Sons, Inc, New York, 1998.

[44] R. Kraut and L. Streeter. Coordination in software development. *CACM*, pages 69–81, 1995.

[45] H. Kunihiko, O. Sheng, B. Shin, and A.J. Figueredo. Understanding relationship among teleworkers' e-mail usage, e-mail richness perception, and e-mail productivity perception under a software engineering environment. *IEEE Transactions on Engineering Management*, 47(2), May 2000.

[46] S. Lewis. Extending eclipse to support cscw. *In proceedings, CSCW Conference, Toronto*, 2004.

[47] S. Lussier. New tricks: How open source changed the way my team works. *IEEE Software*, 21(1):68–72, 2004.

[48] Nazim H. Madhavji. The prism model of changes. In *Proceedings of the 13th International Conference on Software Engineering*, pages 166–177. IEEE Computer Soc. Press., 1991.

[49] A. Martinez, Y. Dimitriadis, B. Rubia, E. Gomez, and P. de la Fuente. Combining qualitative evaluation and social network analysis for the study of classroom social interactions. *Computers & Education*, 41(4):353–368, 2003.

[50] L.J. May. Major causes of software projects failures. *CROSSTALK The Journal of Defense Software Engineering*, July 1998.

[51] A. Mockus, R. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[52] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001.

[53] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the conference on the future of software engineering*, pages 35–46, 2000.

[54] J.S. O'Neal and D.L. Carver. Analyzing the impact of changing requirements. *In Proceedings of the IEEE International Conference on Software Maintenance*, pages 190–195, 2001.

[55] J. Orasanu and E. Salas. Team decision making in complex environments. *In G. Klein, J. Orasanu, and R. Calderwood (Eds.), Desicion Making in Action: Models and Methods. Norwood, NJ: Ablex Publishing Co.*, 1993.

[56] M. Paasivaara and C. Lassenius. Collaboration practices in global inter-organizational software development projects. *Software Process: Improvement and practice*, 8(4):183–200, 2003.

[57] C. Reis, R. Pontin, and M. Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175. C. Gacek and B. Arief., 2002.

[58] A. Sarma, Z. Noroozi, and A. Van der Hoek. Palantir: Raising awareness among configuration management workspaces. In *Proceedings of ICSE 2003*, page 444, May 2003.

[59] A. Sathi, T.E. Morton, and S.F. Roth. Callisto: An intelligent project management system. *AI Magazine. Reprinted in Greif (1988)*, pages 34–52, Winter 1986.

[60] W. Scacchi. Understanding the requirements for developing open source software systems. *In IEEE proceedings - software*, 148(1):24–39, 2002.

[61] Kjeld Schmidt. The problem with awareness. *Computer Supported Cooperative Work*, 11:285–298, 2002.

[62] S. Sim and R.C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering*, pages 361–370, 1998.

[63] V. Sinha, B. Sengupta, and S. Chandra. Enabling collaboration in distributed requirements management. *IEEE Special Issues, Global Software Development, Computer Society Pres, to be published*, 2006.

[64] M. Smith, D. Weiss, P. Wilcox, and R. Dewar. The ophelia traceability layer. In *Cooperarative Methods and Tools for Distributed Software Processes, 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes*, pages 150–161, March 2003.

[65] Ian Sommerville. *Software Engineering.* Addinson-Wesley, 2004.

[66] C. Souza, D. Redmiles, L.T. Cheng, D. Millen, and J. Patterson. Sometimes you need to see through walls - a field study of application programming interfaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 63–71, November 2004.

[67] C. Steinfield, C. Jang, and B. Pfaff. Supporting virtual team collaboration: The teamscope system. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 81–90, 1999.

[68] M. Storey, D. Cubranic, and D.M. German. On the use of visualization to support awareness of human activities in software development. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, May 2005.

[69] S.M. Sutton Jr., D. Heimbigner, and L.J. Osterweil. Language constructs for managing change in process-centered environments. In *Proceedings of the fourth ACM SIGSOFT symposium on Software development environments*, pages 206–217, 1990.

[70] E. Trainer, S. Quirk, C.R.B. de Souza, and D.F Redmiles. Bridging the gap between technical and social dependencies with ariadne. In *Proceedings of the Eclipse Technology eXchange (ETX) Workshop*, pages 26–30, 2005.

[71] S. Wasserman and K. Faust. *Social network analysis: methods and applications.* Cambridge: Cambridge University Press, 1994.

[72] B. Wellman. The place of kinfolk in personal community networks. *Marriage and Family Review, 15*, pages 195–228, 1990.

[73] T. Wolf and A.H. Dutoit. Supporting traceability in distributed software development projects. In *Proceedings of the International Workshop on Distributed Software Development*, pages 121–134, August 2005.

[74] B. Yuwono and D.L. Lee. Wise: A world wide web resource database system. *IEEE Transactions on Knowledge and Data Engineering*, 08(4):548–554, 1996.