DEPENDENCIES IN GEOGRAPHICALLY DISTRIBUTED SOFTWARE DEVELOPMENT: OVERCOMING THE LIMITS OF MODULARITY[1]

Marcelo Cataldo

CMU-ISRI-07-120

December 2007

School of Computer Science
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

[1] This dissertation was supported by the National Science Foundation under Grant No. IIS-0414698, Grant No. IIS-0534656 and Grant No. IGERT 9972762, by the U.S. Army Research Laboratory under Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011, by the Office of Naval Research (ONR N00014-06-1-0921) and by the Air Force Research Lab with Charles River Analytics SC060701.

Keywords: geographically distributed software development, collaborative software development, coordination, software dependencies.

Dedicada a Pei-Chi y a mis padres, Antonio Y Mirta

ACKNOWLEDGEMENTS

I have been very fortunate to work with an outstanding dissertation committee in Kathleen Carley, Jim Herbsleb, Len Bass and David Redmiles. I am particularly indebted to Kathleen and Jim for being the best advisors a student could hope for. I also would like to thank my family for their patience and encouragement, specially, my wife Pei-Chi without whom my life as a doctoral student would have been a lot less enjoyable.

Through out this process, many others helped shape my views and research. Special thanks go to Matthew Bass, Audris Mockus, Jeffrey Reminga, Jeffrey Roberts and Patrick Wagstrom.

ABSTRACT

Geographically distributed software development (GDSD) is becoming pervasive. Hence, the constraints in communication and its negative impact of developers' ability to coordinate effectively is a growing problem that consistently results in sub-par performance of GDSD teams. Past research argues that geographically distributed teams do better when their work is almost independent from each other. In software engineering, modularization is the traditional technique intended to reduce the interdependencies among modules that constitutes a system. The modular design argument suggests that by reducing the technical dependencies, the work dependencies between teams developing interdependent modules are also reduced. Consequently, a modular product structure leads to an equivalent modular task structure. This dissertation argues that modularization is not a sufficient representation of work dependencies in the context of software development and it proposes a method for measuring socio-technical congruence, defined as the relationship between the structure of work dependencies and the coordination patterns of the organization doing the technical work. Two empirical studies assessed the impact of socio-technical congruence on development productivity and product quality. In addition, a third empirical study explores how developers in a geographically distributed software development organization evolve their coordination patterns to overcome the limitations of the modular design approach.

Collectively, this dissertation has important contributions to software engineering, CSCW and organizational literatures. First, the empirical evaluation of the congruence framework showed the importance of understanding the dynamic nature of software development. Identifying the "right" set of product dependencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution time of modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively stable view of product dependencies that is not representative of the dynamism in product dependencies that emerges as software systems are implemented. On the other hand, logical dependencies provide a more accurate representation of the most relevant product dependencies in software development projects. Secondly, this dissertation moves forward our understanding of the relationship between product and work dependencies and software quality. Logical dependencies among software modules and work dependencies were found to be two very significant factors affecting the failure proneness of software modules. Finally, the longitudinal analysis of coordination activities in a GDSD project showed that developers centrally positioned in the social system of information exchanges and coordination activities performed a critical bridging function across formal teams and geographical locations. Moreover, those same individuals contributed an average of 57% of development effort in terms of implementing the software system in each release covered by the data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Over the past couple of decades, geographically distributed work has become pervasive and software development organizations are no exception. Factors such as access to talent, acquisitions and the need to reduce the time-to-market of new products are the driving forces for the increasing number of geographically distributed software development (GDSD) projects (Herbsleb & Moitra, 2001; Karolak, 1998). Unfortunately, this new trend has its costs. Distance leads to numerous problems in communication and coordination, and ultimately, impacts the performance of software development teams (Herbsleb et al, 2000; Herbsleb & Mockus, 2003). The failure to identify work dependencies among developers or development teams results in coordination problems. A growing body of work on coordination in software development suggests that the identification and the management of dependencies is a fundamental challenge in software development organizations, particularly in those that are geographically distributed (some examples are: Cataldo et al, 2007; de Sourza, 2005; Grinter et al, 1999; Herbsleb et al, 2000; Herbsleb & Mockus, 2003). The modular product design literature has developed an important body of research on interdependency, for instance, the work on design structure matrices to find alternative structures that reduce dependencies among the various components of the system (Eppinger et al, 1994; Sullivan et al, 2001). Interdependency is central to organizations and it has also been a perennial research topic in organizational theory (DeSanctis et al, 1999; Staudenmeyer, 1997). Those research streams could inform the design of software development organizations so they are better able to identify and manage work dependencies. However, we first need to understand the assumptions of the different theoretical views and how those assumptions relate to the characteristics of software development tasks.

The Nature of Software Development and Modular Design

The idea of dividing a complex task into smaller manageable units is consistent with the reductionist view (Simon, 1962; von Hippel, 1990) which is well developed in the product development literature (Eppinger et al, 1994). Projects, typically, have a general description of the system's components and their relationships or a more detailed report such as architectural or high-level design document. Managers use the information in those documents to divide the development effort into work items that are assigned to specific development teams minimizing the interdependencies among those teams (Conway, 1968; Eppinger et al, 1994; Sullivan et al, 2001). In the system design literature, it has long been speculated that the structure of a product inevitably resembles the structure of the organization that designs it (Conway, 1968). In Conway's original formulation, he reasoned that coordinating product design decisions requires communication among the engineers making those decisions. If everyone needs to talk to everyone, the communication overhead does not scale well for projects of any size. Therefore, products must be split into components, with limited technical dependencies[2] among them, and each component assigned to a single team. Conway (1968) proposed that the component structure and organizational structure stand in a homomorphic relation, in that more than one component can be assigned to a team, but a component must be assigned to a single team.

[2] The terms "technical dependency" and "product dependency" are used interchangeably through this dissertation.

A similar argument has been proposed in the strategic management literature. Baldwin and Clark (2000, page 90) argued that modularization makes complexity manageable, enables parallel work and tolerates uncertainty. The design decisions are hidden within the modules which communicate through standard interfaces, then, modularization adds value by allowing independent experimentation of modules and substitution (Baldwin & Clark, 2000). Moreover, Baldwin and Clark (2000, page 89) argued that a modular design structure leads to an equivalent modular task structure. Then, their view aligns with Conway's idea that one or more modules can be assigned to one organizational unit and work can be conducted almost independently of others. In the context of software engineering, a similar approach was first articulated by Parnas (1972) as modular software design. Parnas (1972) argued that modules ought to be considered work items instead of just a collection of subprograms. Then, development work can continue independently and in parallel across different modules. Parnas' views also coincide with the theoretical arguments from product design and strategic management literatures.

All three theoretical views rely on two interrelated assumptions. The authors assume a simple and obvious relationship between product modularization and task modularization. Hence, reducing the technical interdependencies among modules, the modularization theories argue, task interdependencies are reduced, which consequently, reduces the need for communication among work groups. Unfortunately, there are several problems with these assumptions. First, existing software modularization approaches only use a subset of the technical dependencies, typically syntactic relationships, of a software system (Garcia et al, 2007). Then, potentially relevant work dependencies might be ignored. Secondly, recent empirical evidence indicates that the relationship between product structure and task structure is not as simple as previously assumed. Moreover, the theorized similarity between product and task structures diminishes over time (Cataldo et al, 2006).

Thirdly, promoting minimal communication between teams responsible for interdependent modules is problematic. The computer-mediated communication literature suggests that loose-coupling tasks is the appropriate approach when teams are geographically distributed (Olson & Olson, 2000). However, recent studies suggest that minimal communication between teams, collocated or distributed, is detrimental to the success of projects. The product development literature argues that information hiding, which leads to minimal communication between teams, is an inevitable antecedent of variability in the evolution of projects resulting, typically, in integration problems (Yassine et al, 2003). In context of software development, de Souza and colleagues (2004) found that information hiding led development teams to be unaware of others teams' work resulting in coordination problems. Grinter and colleagues (1999) reported similar findings for geographically distributed software development projects. The authors highlighted that the main consequence of reducing the teams' need to communicate was to increase costs because problems were discovered too late in the development process. Those findings do not suggest that modularization is not useful. They highlight the need to supplement it with coordination mechanisms to allow developers to deal correctly with the assumptions that are not captured in the specification of the dependencies.

Another problem associated with the assumptions of modular design is the nature and stability of the interfaces between software modules. Although, the program dependency literature defines technical dependencies as a syntactic or semantic relationship between statements (Podgurski & Clarke, 1990), the same ideas are applied at the level of modules. Then, relationships among modules could also range from syntactic, for instance a function call from module A to module B, to more complex semantic dependencies where, for example, the computations done in one module affects the behavior of another module. Some authors refer to those types of semantic dependencies as dynamic (Bass et al, 2003) or logical (Gall et al, 1998). Even in the simple case of a function call between two modules, the complexity and the degree of dependency varies, for instance, if we consider the number of parameters of a function call or we compare parameters passed by value versus parameters passed by reference. Cataldo et al (2007) presented case studies where even simple interfaces between modules developed by remote teams create coordination breakdown and integration problems. The authors reported that semantic dependencies were even more problematic and they argued that the developers' ability to identify and manage dependencies was hindered by several inter-related factors such as development processes, organizational attributes (e.g. structure, management style) and uncertainty of the interfaces. In a field study of a large software project, de Souza (2005) encountered that interfaces tended to change often and their design details tended to be incomplete, leading to serious integration problems. These findings argue that the interfaces between software modules might differ in complexity and, often, it is not possible to specify those interfaces at the necessary level of detail, increasing the likelihood of future changes to them. This lack of stability represents a constant challenge for software development organizations.

In sum, the modularization approach is a very useful tool for dividing the development of a complex software system into manageable units. However, modularization is not a sufficient representation of work dependencies in software development activities. The relationship between the task dependency structure and the product structure is not as simple as theorized. Appropriate mechanisms are then required to identify relevant work dependencies and, consequently, maintain suitable levels of communication and coordination among teams developing interdependent modules, particularly, in the case of geographically distributed software development.

The Nature of Software Development and Interdependency Theories

Coordination is a central concept in organizations, the idea of division of labor into interdependent units is a well developed and mechanisms for coping with the varying degree of interdependency have been proposed in the traditional organizational literature (for instance, March & Simon, 1958; Thompson, 1967; Galbraith, 1973; Staudenmayer, 1997). More recent work, particularly in organizational design, has focused on computational and mathematical approaches to examine how organizational designs, that use different models of communication and coordination, are affected by factors such as stress, task decomposition, quality of information exchanged, and ability to adapt (for instance, Carley and Lin, 1995, 1997; Handley & Levis, 2001; Perdu & Levis, 1998). Then both streams of work, traditional organizational theory and computational and mathematical organizational theory (CMOT), are relevant to the problem of coordination in software development projects.

In the traditional organizational theory, March and Simon (1958) argued that coordination encompasses more than just a traditional division of labor and assignment of tasks. The authors proposed numerous mechanisms such the division of the task into nearly independent parts and they also argued that schedules and feedback mechanisms are required when interdependence is unavoidable. Thompson (1967) extended March and Simon's work by matching three mechanisms: standardization, plan, and mutual adjustment, to stylized categorizations of dependencies such as pooled, sequential, and reciprocal. Galbraith (1973) argued that low levels of interdependency can be managed by traditional mechanisms such as rules and programs. However, as the level of interdependency increases additional mechanisms are required such as slack resources and lateral communication (Galbraith, 1973). Mintzberg (1979) took an organizational-level perspective and argued that specific coordination mechanisms are properties of particular kinds of organizations and environments. Crowston (1991) developed a typology of coordination problems to catalog coordination mechanisms that address specific types of interdependencies. Staudenmayer (1997) grouped the contributions of March and Simon, Thompson, and others into the information processing theories of interdependency which, she argued, rely on the assumptions of determinism and stability. In other words, those theoretical views focus on predictable and static tasks (Staudenmayer, 1997). This limitation of the information processing argument is not problematic if software development tasks can be identified a priori and the set of interdependencies that arise from the division of labor are managed with the appropriate set of mechanisms. If we think in terms of project management activities, coarse-grain development activities such as "develop component A" or "implement feature X" can typically be identify at relatively early stages of the projects. Some dependencies among those development tasks are typically easy to identify. For instance, particular work items need to be finished before other work items can start. Work items that can only be assigned to specific teams because of the skill set required would represent another example. Then, specific organizational forms can be used to manage the dependencies among those coarse-grain development tasks (Malone & Crowston, 1991), even in the case of geographically distributed development organizations (Grinter et al, 1999).

Unfortunately, there are several characteristics of software development activities that limit the applicability of traditional organizational theories as well as the more recent CMOT work. First, it is widely accepted among software engineering researchers and practitioners that the requirements of the system become known over time or those requirements change as time progresses (Leffingwell & Widrig, 2003). In some cases the changes in the requirements result in minor alterations of specific development tasks. In other cases, new features have to be added or features under development are eliminated. These events introduce a certain level of dynamism in software development that challenges the determinism and stability assumptions of the information processing views of interdependency.

Secondly, the dynamic nature of finer-grain dependencies that arise as part of the development of a piece of code is not well suited for traditional organizational theories of coordination. The act of developing a software system consists of a collection of design decisions, either at the architectural level or at the implementation level. Those design decisions introduce constraints that might establish new dependencies among the various parts of the system, modify existing ones or even eliminate dependencies. The changes in dependencies can generate new coordination requirements that are quite difficult to identify a priori, particularly when they are not obvious, or as a project matures over time (Henderson & Clark, 1990; Sosa et al, 2004). Failure to discover the changes in coordination needs might have a profound impact on the quality of the product (Curtis et al, 1988), on productivity (Herbsleb & Mockus, 2003) and even on the projects' overall design (Bass et al, 2006). In addition, little is known about the specific impact of the various types of dependencies that arise among parts of a software system such as explicit versus implicit dependencies or syntactic versus logical dependencies. Then, the use of the computational and mathematical organizational theory approaches is limited because of the lack of theoretical framework that guides the modeling of the relationships between the organizational tasks, their dependencies and the need to communicate and coordination.

In sum, software development tasks are embedded in an evolving network of coordination requirements that need to be satisfied. The coarse-grain and idealized approaches suggested by the organization theory literature are not appropriate to identify and manage such a dynamic web of interdependencies. A finer-grain view of coordination would provide a better framework in dynamic knowledge-intensive tasks such as software development.

Research Questions

In the previous sections, I highlighted the limitations of the current mechanisms for identifying and managing dependencies in geographically distributed software development organizations. Product modularization does not necessarily yield an equivalent task modularization structure and additional mechanisms are required to maintain appropriate levels of coordination among workgroups. The nature of software development such as the attributes and stability of interfaces among modules and the dynamics of technical dependencies, limit the applicability of established task decomposability and coordination approaches. Moreover, these characteristics are a constant challenge for software development organizations, particularly, for those geographically distributed. This dissertation addresses the problem of work dependencies in software development by examining how to use technical dependencies to determine work dependencies and by investigating the impact of those work dependencies in the development process. Specifically, I address the following general research questions:

RQ 1: How relevant task dependencies can be identified from technical dependencies?

RQ 2: What is the impact of those task dependencies on traditional outcome variables such as productivity and quality?

The rest of this document is organized as follows. Chapter 2 presents a framework for identifying and managing dependencies. Chapter 3 introduces terminology used in this dissertation and describes the various datasets used in the empirical studies. In chapter 4, I examine different methods of identifying work dependencies from technical dependencies. Chapter 5 presents the first empirical study that examines the impact on development productivity of the mismatches between coordination requirements and coordination behavior. In chapter 6, I study the impact of the structure of technical and work dependencies on software quality. The last empirical study which explores the usage of the proposed framework for examining the relationship between coordination behavior and developer-level performance is described in chapter 7. Chapter 8 describes developer and managerial applications of the results reported in this dissertation. Finally, chapter 9 describes the contributions of this research endeavor, its limitations as well as future research directions.

CHAPTER 2: A FRAMEWORK FOR IDENTIFICATION OF WORK DEPENDENCIES

It has long been observed that organizations carry out complex tasks by dividing them into smaller interdependent work units assigned to groups and coordination arises as a response to those interdependent activities (March & Simon, 1958). Communication channels emerge in the formal and informal organizations. Over time, those information conduits develop around the interactions that are most critical to the organization's main task (Galbraith, 1973). This is particularly important in product development organizations which organize themselves around their products' architectures because the main components of their products define the organization's key subtasks (von Hippel, 1990). Organizations also develop filters that identify the most relevant information pertinent to the task at hand (Daft & Weick, 1990). Changes in task dependencies, however, jeopardize the appropriateness of the information flows and filters and can disrupt the organization's ability to coordinate effectively. For example, Henderson & Clark (1990) found that minor changes in product architecture can generate substantial changes in task dependencies, and can have drastic consequences for the organizations' ability to coordinate work. If effective ways of identifying detailed work dependencies and tracking their changes over time exist, we would be in a much better position to design mechanisms that could help to align information flow with work dependencies.

Identifying work dependencies and determining the appropriate coordination mechanism to address the dependencies is not a trivial problem. Coordination is a recurrent topic in the organizational theory literature and many stylized types of task dependencies and coordination mechanisms have been proposed over the past several decades (Crowston, 1991; Galbraith, 1973; Malone & Crowston, 1994; March & Simon, 1958; Mitzberg, 1979; Thompson, 1968). However, numerous types of work, in particular non-routine knowledge-intensive activities, are potentially full of fine-grain dependencies that might change on a daily or hourly basis. Conventional coordination mechanisms like standard operating procedures or routines would have very limited applicability in these dynamic contexts. Therefore, designing mechanisms to handle rapidly shifting coordination needs requires a more fine-grained level of analysis than what the traditional views of coordination provide.

In the context of software development, a technical dependency in the software system represents a coordination need that relevant software developers might need to address. The result of ignoring coordination requirement could lead to increased number of defects, problems in integration and longer development time (Curtis et al, 1988; Espinosa et al, 2002; Kraut et al, 1995; Herbsleb & Mockus, 2003). When members of a team are physically collocated and coordination requirements involve individuals from the same team, there are numerous ways for team members to identify the needs to coordinate and act on them such as group and status meetings and managerial intervention. The problem of identifying the need to coordinate is further complicated when coordination requirements change rapidly (Cataldo et al, 2006). In this chapter, I present a framework to determine the coordination requirements among developers. The objective of the framework is two-fold. First, provide a fine-grain level of analysis of coordination. The second objective is to allow for identification of work dependencies from alternative representations of technical dependencies of the system. I also propose a measure of “fit” between work dependencies and the coordination activities performed by the software developers.

The Concept of Socio-Technical Congruence

Product development endeavors involve two fundamental elements: a technical and a social component. The technical properties of the product to develop, the processes, the tasks, and the technology employed in the development effort constitute the technical component. The second element is composed by the organizational individuals involved in the development process, their attitudes and behaviors. In other words, a product development project can be thought of a socio-technical system where the two components, the technical and the social elements, need to be aligned in order to have a successful project. Then, a key issue is to understand how we can examine the relationship between those two, the technical and the social, dimensions. Two lines of work are particularly relevant in this context. First, the concept of “fit” from organizational literature refers to the match between a particular organizational design and the organization’s ability to carry out a task (Burton & Obel, 1998). The work in this line of research has, traditionally, focused on two factors: the temporal dependencies among tasks that are assigned to organizational groups and the formal organizational structure as a means of communication and coordination (Carley & Ren, 2001; Levchuck et al, 2004). Secondly, the research on dynamic analysis of social networks provides an innovative approach, called the meta-matrix, to examine the dynamic co-evolution of relationships among multiple types of entities such as resources, tasks, and individuals (Carley, 2002; Krackhardt & Carley, 1998). The concept of socio-technical congruence presented in this chapter builds on the idea of "fit" from the organizational theory literature and from a mathematical stand point builds on the meta-matrix model from the dynamic network analysis literature. Combining those two lines of research allows for two important contributions to the literature. First, the socio-technical congruence framework presented here provides a fine-grain level of analysis. Secondly, the measure facilitates assessing the role of coordination activities in multiple and complementary ways as well as examining the impact of several types of dependencies.

Figure 1 presents an intuitive representation of the measure of congruence formally defined later in this chapter. A group of workers have a set of work dependencies which defines a set of coordination requirements. When the coordination activities carried out by those workers define a pattern of coordination similar to those defined by the coordination requirement (case A in Figure 1), we have high levels of congruence or "good fit". If the patterns of coordination requirements and coordination activities do not match, we have low levels of congruence or a "poor fit" (case B in Figure 1).

Formally, socio-technical congruence is defined as the match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by the workers. In other words, the concept of congruence has two components, coordination needs and coordination activities, and the following sections discuss the mathematical framework to measure them.

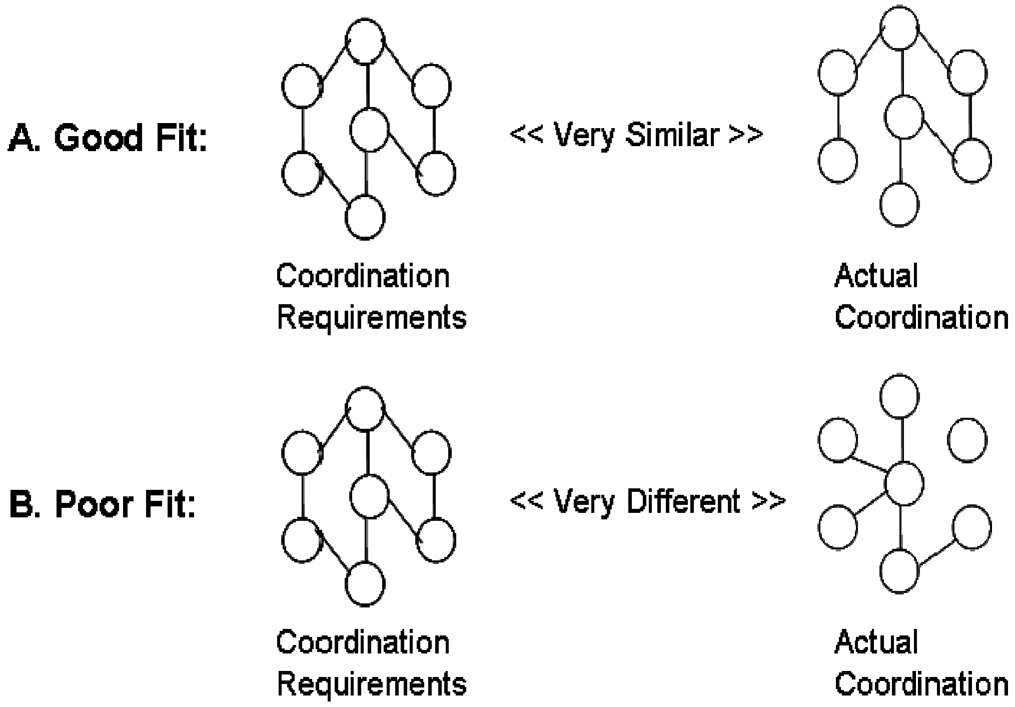

Figure 1: The Concept of Congruence

Identification of Coordination Requirements

In order to identify which set of individuals should be coordinating their activities, we need to represent two sets of relationships. One set is represented by which individuals are working on which tasks. The relationships or dependencies among tasks represent the second element. Past research has used a matrix formalization to capture and relate those two pieces of information. For instance, Carley and Ren (2001) proposed a metric, called resource congruence, to measure the relationship between the resources required to perform a task and workers' access to those resources. The same metric was further examined by Carley and colleagues (2003) in the context of covert networks.

In the framework proposed in this chapter, assignments of individuals to particular work items is be represented by a people by task matrix where a one in cell *ij* indicates that worker *i* is assigned to task *j*. I will refer to this matrix as *Task Assignments* (T_A). Following the same approach, the set of dependencies among tasks can be represented as a square matrix where a cell *ij* (or cell *ji*) indicates that task *i* and task *j* are interdependent. I will refer to this matrix as *Task Dependencies* (T_D). Now, if the *Task Assignment* and *Task Dependencies* matrices are multiplied, a people by task matrix is obtained that represents the set of tasks a particular worker should be aware of, given the work items the person is responsible for and the dependencies of those work items with other tasks. Finally, a representation of the coordination requirements among the different workers is obtained by multiplying the product of the *Task Assignment* and *Task Dependencies* matrices by the transpose of the *Task Assignment* matrix. This product results in a people by people matrix where a cell *ij* (or cell *ji*) indicates the extent to which person *i* works on tasks that share dependencies with the tasks worked on by person *j*. In other words, the resulting matrix represents the *Coordination Requirements* or the extent to which each pair of people needs to coordinate their work. Formally, the *Coordination Requirements* matrix is determined by the following product:

$$\mathbf{C_R = T_A * T_D * T_A^T} \qquad \textbf{(Equation 1)}$$

where, T_A is the Task Assignments matrix, T_D is the Task Dependencies matrix and T_A^T is the transpose of the Task Assignments matrix.

This framework provides alternatives ways of thinking about coordination requirements among workers depending on what type of data is used to populate the Task Dependencies matrix. Past work had focused on temporal relationships between tasks, for instance, task A needs to be done before task B (e.g. Levchuk et al, 2003). In the context of software development, such way of thinking about task dependencies is quite common. Alternative views could be based on high level roles in the development organizations (e.g. integration and testing depends on development) or task dependencies based on product dependencies in the actual software code (e.g. function calls between modules). The focus on this dissertation is on the work dependencies structure-product dependency structure relationship because, as argued in chapter 1, the difficulty of identifying and managing certain types of product dependencies is a critical factor in coordination success and ultimately in productivity and quality.

Measuring Socio-Technical Congruence

Given a particular *Coordination Requirements* matrix constructed from relating product dependencies to work dependencies, we can compare it to an *Actual Coordination* (C_A) matrix that represents the interactions workers engaged in through different means of coordination. I refer to the match between those to matrices as socio-technical congruence. Then, given a particular set of dependencies among tasks, congruence is the proportion of coordination activities that actually occurred (given by the *Actual Coordination* matrix) relative to the total number of coordination activities that should have taken place (given by the *Coordination Requirements* matrix). For example, if the *Coordination Requirements* matrix shows that 10 pairs should coordinate, and of these, 5 show *Actual Coordination* interactions, then the congruence is 0.5. Formally, we define congruence as follows:

$$\text{Diff}(C_R, C_A) = \text{card}\ \{\ \text{diff}_{ij} \mid cr_{ij} > 0\ \&\ ca_{ij} > 0\ \}$$

$$|C_R| = \text{card}\ \{\ cr_{ij} > 0\ \}$$

We have,

$$\textbf{Congruence}\ (C_R, C_A) = \textbf{Diff}\ (C_R, C_A)\ /\ |C_R| \qquad \textbf{(Equation 2)}$$

In sum, the value of congruence belongs to the [0,1] interval that represents the proportion of coordination requirements that were satisfied through some type of coordination activity or mechanism. The measure of socio-technical congruence proposed here provides a new way of thinking about coordination, particularly, by providing a fine-grain level of analysis of different types of product dependencies and allowing us to examine how coordination needs are impacted by them.

CHAPTER 3: TERMINOLOGY AND DESCRIPTION OF THE DATASETS

Terminology

In this section, I define several terms are used through out the empirical studies as well as the description of the datasets:

Source code file: A source code file represents a collection of functions, methods, and data type declarations and definitions that implement part of or an entire functionality of a software system. In this dissertation, I will use the terms source code file and module interchangeably. This definition does not refer or imply any specific way of partitioning a system into implementation modules.

Commit: A commit represents an actual modification to one or more source code files in the version control system. A particular commit contain at least the following attributes: a date of submission, an author or developer responsible, a list of one or more files and the modifications to those files. The terms submission and changelist are used as synonyms of a commit through out this document.

Modification request (MR): A modification request represents a work item that refers to a conceptual change to the software that involves modifications to a set of source code files (Mockus & Weiss, 2000). The changes could represent the development of new functionality or the resolution of a defect encountered by a developer, the quality assurance organization or reported by a customer. A modification request consists of one or more commits from a version control system.

Lines of code (LOC): In various parts of the dissertation, we refer to lines of code as a measure of size of a system or a module. The measure refers to non-blank non-comment lines of code.

Datasets

In order to address the research questions outlined in chapter 1, data from several geographically distributed software development projects was collected. The characteristics of those projects and the data are described in the rest of this chapter.

Project A

I collected data from a software development project of a large distributed system produced by a company that operates in the data storage industry. The data covered a period of 39 months of development activity and the first four releases of the product. The company had one hundred and fourteen developers grouped into eight development teams distributed across three development locations. All the developers worked full time on the project during the time period covered by the data. The system was composed of about 5 million lines of code distributed in 7737 source code files mostly in C language and a small portion (117 files and less than 96000 lines of code) in C++ language. The data corresponding to a total of 8,257 resolved modification requests were identified. Those MRs involved 67,652 commits to the version control system.

Software developers communicated and coordinated using various means. Opportunities for interaction exist when working in the same formal team or when working in the same location. Developers also use tools such as Internet Relay Chat (IRC) and a MR tracking system to interact and coordinate their work. For instance, the MR tracking system keeps track of the progress of the task, comments and observations made by developers as well as additional material used in the development process. I collected communication and coordination information from these two systems. Finally, I also collected demographic data about the developers such as their programming and domain experience and level of formal education.

Project A represents the main source of data for the various empirical studies presented in this dissertation. In order to address potential external validity concerns, data from additional projects was used in each empirical study. Those projects are described in the following paragraphs.

Project B

Version control data from three open source projects from the Apache Software Foundation was collected. I focused on changes to the software that were associated with a modification request that were resolved between February of 2001 and January of 2003. There were a total of 1068 modification requests resolved in that timeframe involving 1972 commits in the version control system. Those modification requests were related to three different projects, Ants, Tomcat and Structs, where a total of seventy five engineers participated in the development effort.

Project C

The project involved the development of an embedded software system for a communications device developed by a major telecommunications company. Forty engineers participated in the project. The data covered a period of five years and the last six releases of the product. All the developers but one worked in the same development facility located in the United States. The remote developer worked in Australia. The system was composed of approximately 1.2 million lines of C and C++ code distributed in 1224 modules with 427 modules written using in C++ language. Data associated with about 7000 modification requests constituted the dataset.

Project D

This project was a large medical device system where the development organization had eighty three engineers grouped into 10 teams distributed across for development locations, one in India, one in Eastern Europe and two in the United States. Architects, some of the technical leads and managers were also in the development facilities located in the United States. All the developers worked full time on the project during the time period covered by the data. Engineers had formal roles such as architect, team lead, tester or developer. The project was organized into iterations which constitute fixed periods of time, about 8 weeks, focused on the development of a set of requirements defined at the beginning of the iteration. The data covered the 7th iteration of the project. A survey instrument based on a roster approach was used to collect coordination activity twice during the development iteration.

CHAPTER 4: METHODS FOR IDENTIFYING WORK DEPENDENCIES IN SOFTWARE DEVELOPMENT PROJECTS

In this chapter, I explore different methods of determining work dependencies from product dependencies (e.g. relationships among the source code files of a software system). Then, those work dependencies will allow us to identify coordination requirements among software developers as proposed in the congruence framework introduced in chapter 2.

Two Approaches to Determine Product Dependencies in Software Systems

The traditional view of software dependency has its origins in compiler optimizations and they focus on control and dataflow relationships (Horwitz et al, 1990). This approach extracts relational information between specific units of analysis such as statements, functions or methods, as well as modules, typically, from the source code of a system or from an intermediate representation of the software code such as bytecodes or abstract syntax trees. These relationships can represent either a data-related dependency (e.g. a particular data structure modified by a function and used in another function) or a functional dependency (e.g. method A calls method B). This type of dependency analysis techniques has been widely used in a research context to examine the relationship between coupling and quality of a software system (e.g. Hutchins & Basili, 1985; Selby & Basili, 1991). Syntactic dependency analysis are also used by software developers to improve their understanding of programs and the linkages among the various parts of those programs (Murphy et al, 1998).

One characteristic of these relational structures such as a call-graph, and for that matter other graphs such as inheritance and data dependencies graphs, is that they provide a particular view of the system-wide structure. Moreover, the accuracy of the information represented in these graphs depends on the ability of the tool used to identify all the appropriate types of syntactic relationships allowed by the underlying programming language (Murphy et al, 1998).

An alternative mechanism of identifying dependencies consists of examining the set of source code files that are modified together as part of a modification request. This approach is equivalent to the approach proposed by Gall and colleagues (1998) in the software evolution literature to identify logical dependencies between modules. A source code file can be viewed as representing a "bundle" of technical decisions. If a modification request can be implemented by changing only one file, it provides no evidence of any dependencies among files. However, when a modification request requires changes to more than one file, it can be assumed that decisions about the change to one file in a modification request depend in some way on the decisions made about changes to the other files involved in implementing the modification request. Dependencies could range from syntactic, for instance a function call between files, to more complex semantic dependencies where the computations done in one files affects the behavior of another files. This approach would represent a better estimate for semantic dependencies relative to call graphs or data graphs because it does not rely on language constructs to establish the dependency relationship between source code files. The remainder of this dissertation refers to this approach to identify dependencies as the "Files Changed Together" (FCT) method. I will refer to the method to identify dependencies based on syntactic functional and data relationship described earlier as the CGRAPH method.

The *Task Dependency* (T_D) matrices produced by the techniques described in the previous paragraphs could change over time as new product dependencies are created or existing ones are removed. Moreover, the information captured by the T_D matrix constructed with the FCT method might differ from the T_D matrix constructed with the CGRAPG method. Those changes or differences could potentially impact the measures of coordination requirements (equation 1) and congruence (equation 2). Then, understanding the general properties of the task dependency matrices, how they evolve over time and how the differ from each other is critical to assess the impact of socio-technical congruence on outcome variables such as development productivity and software quality. The following sections address these issues using the data from Project A.

General Properties and Evolution of the FCT Task Dependency Matrix

Using the FCT method, I constructed monthly T_D matrices which captured all the changes to the code associated with the set of modifications resolved on each month. Since a graph and a matrix are equivalent representations of a set of relational data, I can use widely accepted graph measure to examine the general properties of the T_D matrices[3]. One basic measure is the density of the graph which provides a general idea of the level of interconnectivity among the nodes of the graph. In this research context, density translates to the overall degree of interdependence amongst the source code files in the

[3] I use the terms graph and network interchangeably throughout the dissertation

system. A second useful network measure is the clustering coefficient (Watts, 1999) and indicates the extent to which there are clusters of interdependent source code files that are also interdependent amongst themselves. Those two measures, density and clustering coefficient, provide a general view of the structural properties of the T_D matrices.

Figure 2 shows the evolution of the density and clustering coefficient measures over the time covered by the data. The density of the monthly T_D matrices is relatively low, with a few exceptions where the levels of density exceed 0.01 (avg=0.0033, min=0.0004, max=0.0204). The clustering coefficient measure shows modest levels (avg=0.0925, min=0.0023, max=0.1774) suggesting a small degree of interdependent clusters of files in the T_D matrices. In sum, the results indicate that, on a monthly basis, a small set of dependencies are identified, and those dependencies tend to be modestly clustered.

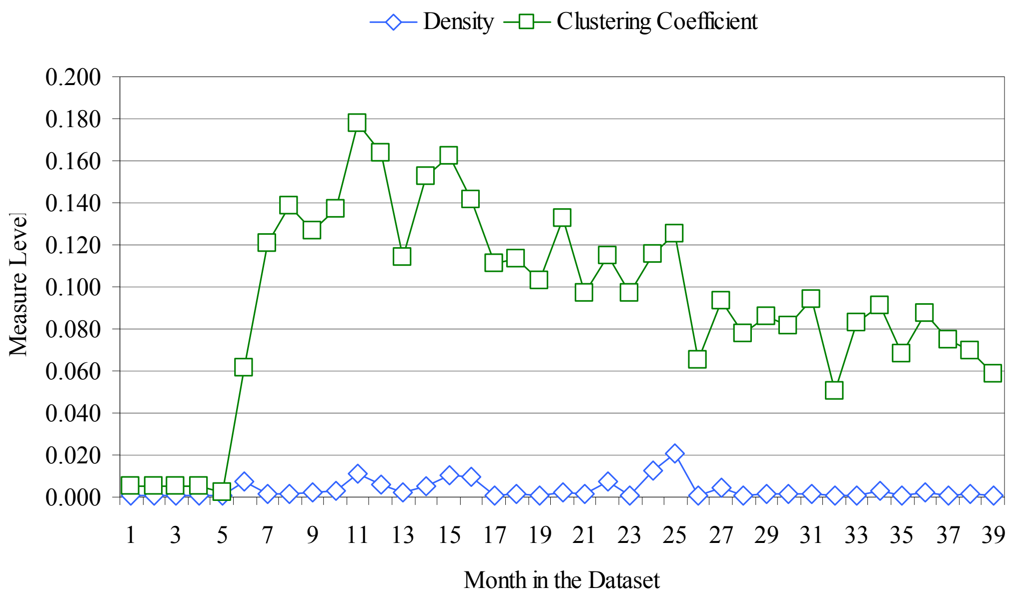

Figure 2: Evolution of the Density and Clustering level of the T_D matrices (FCT method)

An instance of a set of source code files changing together as part of a modification request represents a piece of evidence indicating the existence of a product dependency, potentially logical or implicit in nature. In order to capture the representative set of product dependencies, an understanding of the degree of change in the information contained in the T_D matrices is required. If the matrices are relatively stable that suggests that considering a short time slice could suffice to capture all relevant product dependencies. On the other hand, if the information contained in the monthly T_D matrices changes significantly from time *t* to time *t+1*, it is necessary to identify the appropriate time window size that would yield an accurate representation of the product dependencies. Figure 3 shows the percentage of change in the information contained in a T_D matrix from time *t* relative to the T_D matrix from time *t-1*. The set of technical dependencies captured differ significantly from month to month with an average change of 37% (min=5.11%, max=49.94%). These results suggest that the changes to the source code are affecting different sets of source code files over time. Hence, it is necessary to explore how many months of information would constitute an accurate and representative set of technical dependencies that could be used to compute the *Coordination Requirement* matrices.

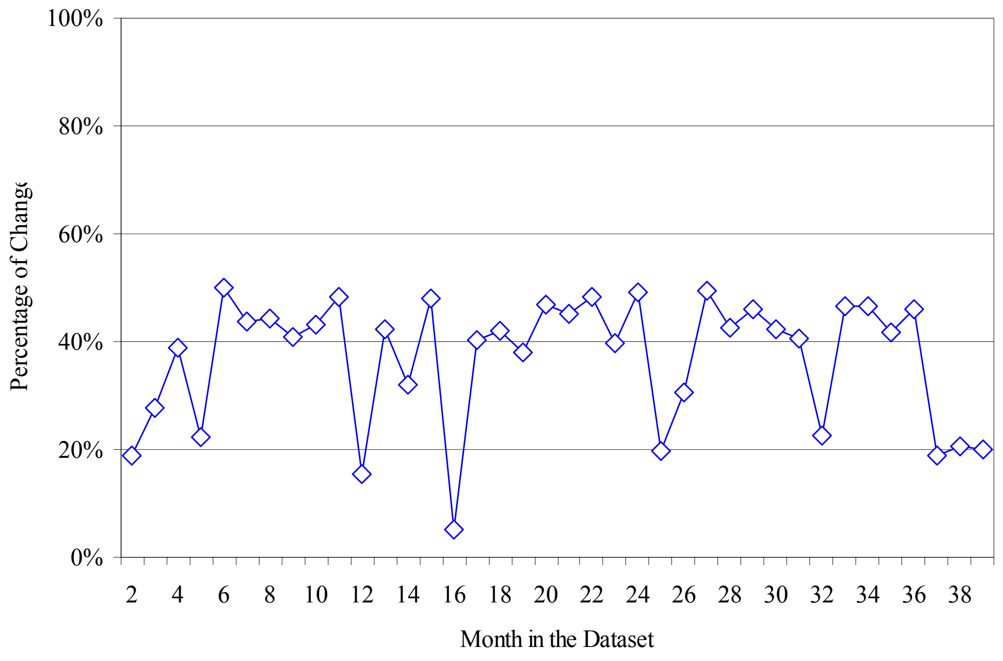

Figure 3: Evolution of the Change in the Information Contained in the T_D matrices (FCT method)

The following procedure was used to explore the time window size necessary to capture the relevant product dependencies. First, the union of all the *k-tuples* of consecutive T_D matrices is computed, where *k* represents the number of months of data used to compute the new T_D matrices and it ranges from 2 to 39 months. For instance, in the case of *k=2*, this computation outputs T_D matrices that contain all the dependencies based on the changes made to the software between months 1 and 2, month 2 and 3, months 3 and 4, and so forth. The second step is to average the network density value of all the matrices associated with a particular value of *k*. Finally, I plotted that average value of network density for each value of *k*. Figure 4 depicts the results of this procedure. As the number of months of data considered to compute the T_D matrix increases, the density level of that T_D matrix increases monotonically until month 19 where a density value of 0.0109 is reached. The remaining 20 months of data increase the density of the T_D matrix from 0.0109 up to 0.01151. In other words, any additional month of data beyond 19 month does not yield a significant increase in the value of the density of the T_D matrix, indicating that any additional month of data does not contribute any additional information value in terms of technical dependencies. In view of this result, I used a time period of 19 months to compute the T_D matrix used in the calculations of the coordination requirements.

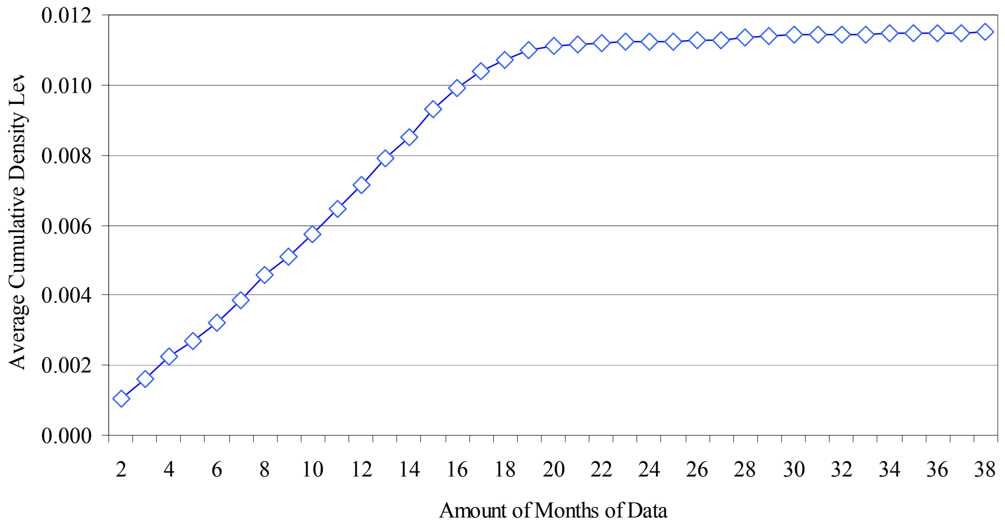

Figure 4: Average Cumulative Density of the T_D matrix (FCT method)

General Properties and Evolution of the CGRAPH Task Dependency Matrix

In this case of the CGRAPH, the dependencies between source code files are determined based on data and functional references. Data references are represented by relationships were a source code file, A, references a data object in a second source code file B. Functional references are represented by relationships where a source code file, A, invokes a function or a method declared in a second source code file B. Unlike the relationships in the FCT methods, data and functional references are directional, that is, the pair of source code files (A,B) is considered different from the pair (B,A).

I collected quarterly data for this type of dependency information, mapping each quarter to the corresponding 3 months of the data discussed in the previous paragraphs. I used the C-REX tool (Hassan and Holt, 2004) to identify programming language tokens and references in each entity of each source code file. This analysis was performed over the entire source code of the system[4] at the end of the 3rd month of each quarter. Using the resulting data, I computed dependencies between source code files by identifying data, function and method references that cross the boundary of each source code file. In other words, each cell *ij* of the T_D matrix computed with the CGRAPH method represents the number of data/function/method references that exist from file *i* to file *j*.

Figure 5 shows the evolution of the network density measure over each quarter. The T_D matrices have higher levels of density (avg=0.0311, min=0.0261, max=0.0322) relative to those obtained using the FCT method[5]. In terms of the evolution of the clustering coefficient measure, we see that the level are also very stable over time, and higher (avg=0.1862, min=0.1738, max=0.1909) than those reported for the T_D matrices created with the FCT method. The density of the T_D matrices produced by the CGRAPH is significantly higher than the density of the matrices produced by the FCT method. This difference could stem primarily from two characteristics of the source code of a system. First, the CGRAPH method identifies numerous technical dependencies that involve files

[4] The set of files used in the analysis also included the automatically generated source code files from functionality such as remote procedure calls.
[5] The maximum level of density of a TD matrix produced by the FCT is 0.01151 if all 39 months of development activity are considered.

that once developed, are rarely modified. Cross-cutting concerns such as logging, tracing and security are good examples. Commonly used low level functionality such memory and thread management and basic storage types such as lists and queues are another example. A second factor that might contribute to higher levels of density of the T_D matrices is the technical dependencies that exist with and between automatically generated source code files. One such example is the source code for remote procedure calls (RPCs). The FCT method would capture dependencies between caller and callee of an RPC if there changes to the RPC specification or functionality. On the other hand, the CGRAPH method would capture the complete path of dependencies from the caller through the RPC stubs, marshalling and communication code all the way to the callee. Given the potential bias that these two factors could have in the computations of dependencies, I removed them from the quarterly call graphs and recomputed the density measures for each quarterly T_D matrices. The results showed a reduction in the density (avg=0.0289, min=0.0241, max=0.0299). However, the density levels remained significantly higher than those for T_D matrices created with the FCT method when considering the 19 month window for development activity.

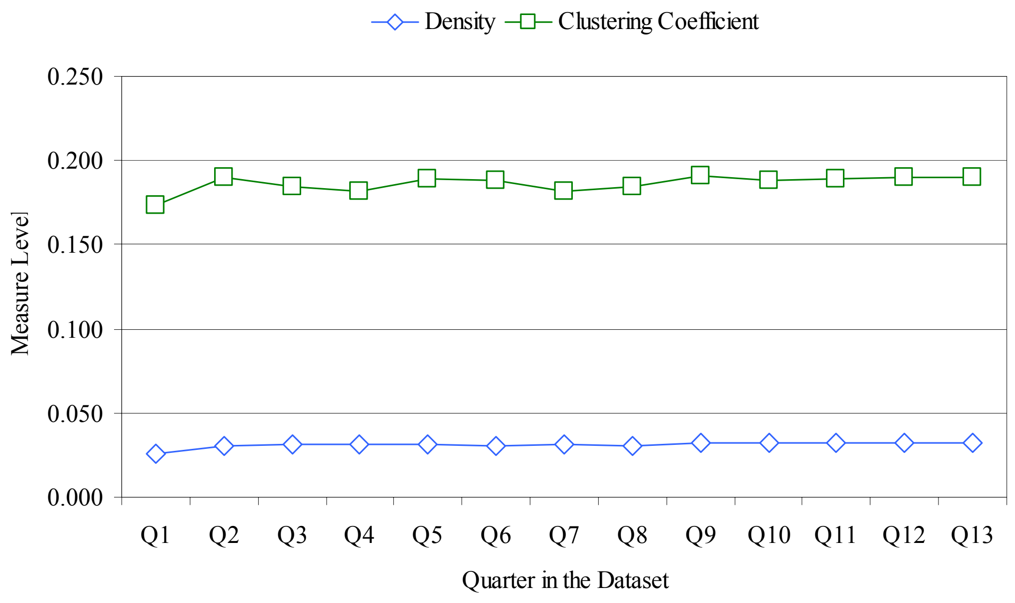

Figure 5: Evolution of the Density level of the T_D matrices (CGRAPH method)

We also examined the percentage of change in the information contained in a T_D matrix from quarter *t* relative to the T_D matrix from quarter *t-1*. Figure 6 shows that rate of change is relatively low (avg=0.24%, min=0.1%, max=0.9%). Those rates of change indicate whether the relationship between files exists or not. If we extend the idea of change to also consider a modification in the weight of the relationship (e.g. number of calls between files), the rate of change increases (avg=1.1%, min=0.4%, max=3%), however, they remain relatively stable over time. This result it is not particularly surprising since significant changes in the overall syntactic dependency structure of a system would imply major code refactoring efforts or architectural changes, events that do not occur often. A similar pattern of stability was found in the T_D matrices produced by the FCT method when I accumulated the commit information from 19 consecutive months. Then, we could think of the volatility that the monthly T_D matrices produced by the FCT method showed as an indication of how the development work evolves over time rather than just focusing how the overall structure of the technical dependencies changes over time. In sum, the CGRAPH method produces T_D matrices that contain significantly more product dependency information relative to those produced by the FCT method. Moreover, a fraction of the product dependencies identified by both methods identified differed significantly.

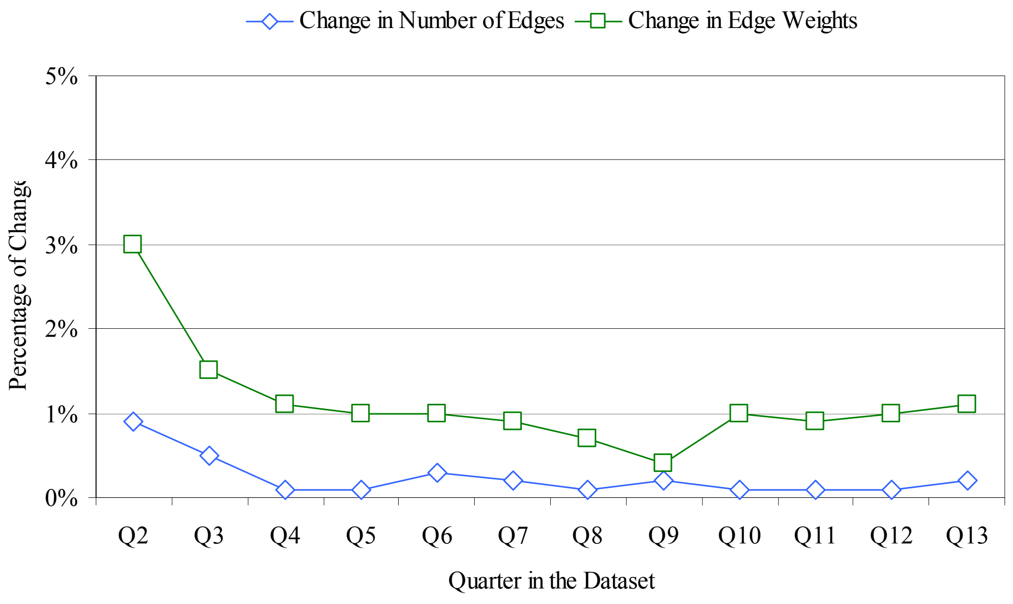

Figure 6: Evolution of the Change in the Information Contained in the T_D matrices (CGRAPH method)

Comparative Analysis of the Task Dependency Matrices

Although the analyses described above provides valuable information about the various T_D matrices, they do not tell us anything regarding the similarity in the sets of technical dependencies identified by both, FCT and CGRAPH, methods. One of the advantages of the FCT method is the potential to identify technical dependencies that might not necessarily be captured by a simple syntactic dependency among modules of a software system such as semantic dependencies (Gall et al, 1998). This argument suggests that a comparison between the T_D matrices generated by the two methods, FCT and CGRAPH, might show differences, possibly significant. The first step of this analysis was to compute the following two operations: $T_D^{(FCT)}$ - $T_D^{(CGRAPH)}$ and $T_D^{(CGRAPH)}$ - $T_D^{(FCT)}$. These operations, which are equivalent to the set difference operation, allow us to determine which dependencies that are identified by the FCT methods are not identified by the CGRAPH method and vice versa. The focus is to identify whether a relationship between two modules exists on one matrix, the other or in both. Hence, I do not consider the differences in the weight on the linkages. I compared quarterly $T_D^{(CGRAPH)}$ matrices against the $T_D^{(FCT)}$ computed for a period of time of the 19 months prior to the end of the quarter. For the first two quarters, I did not have 19 month worth of past data to compute the $T_D^{(FCT)}$ matrices. Therefore, I used 13 months to construct the $T_D^{(FCT)}$ that compared to the $T_D^{(CGRAPH)}$ matrix from the first quarter, and 16 months in the case of the second quarter comparison.

Figure 7 shows the comparison between the T_D matrices. The T_D matrix computed using the FCT method has an average of 14.6% of the dependencies that were not identified by the CGRAPH methods (min=12.4%, max=17.1%). As discussed earlier, the T_D matrices computed using the CGRAPH method are denser and that situation is clearly reflected in this comparison. On average, the T_D matrix computed using the CGRAPH had 74.3% of product dependencies that were not identified by the FCT method (min=70.6%, max=79.2%).

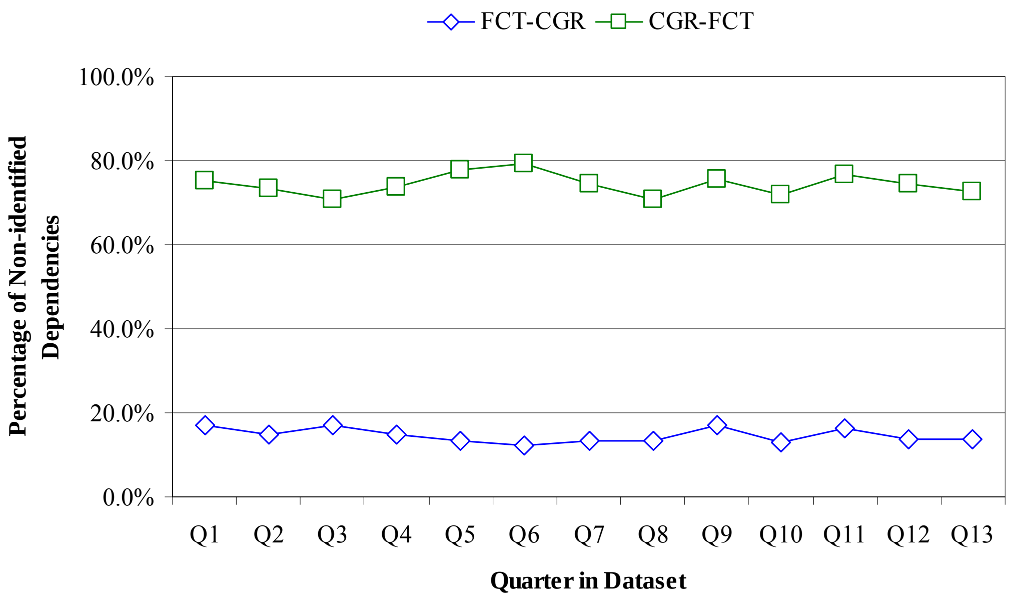

Figure 7: Comparison between T_D matrices generated by the FCT and CGRAPH methods

Comparative Analysis of the Coordination Requirement Matrices

As described in chapter 2, the Coordination Requirements matrix (C_R) is a function of two elements: the T_A matrix and the T_D matrix. Using the different methods for identifying technical dependencies to construct T_D matrices will result in different C_R matrices. Hence, we also need to examine the general properties of the both types of C_R matrices. Using the data from the modification requests resolved in each month to compute the T_A matrix. In terms of computing the T_D matrix, we use a 19 month moving windows in the case of the FCT method or the corresponding quarterly T_D matrix in the case of the CGRAPH method. Figure 8 shows the evolution of the density and clustering coefficient measures for the C_R matrices constructed based on the FCT method. We observe that the density of the monthly C_R matrices is low (avg=0.0655, min=0.0005, max=0.1429) while the clustering coefficient measure shows relatively high levels (avg=0.3179, min=0.0308, max=0.4331) suggesting an important degree of interdependent clusters of files in the C_R matrices.

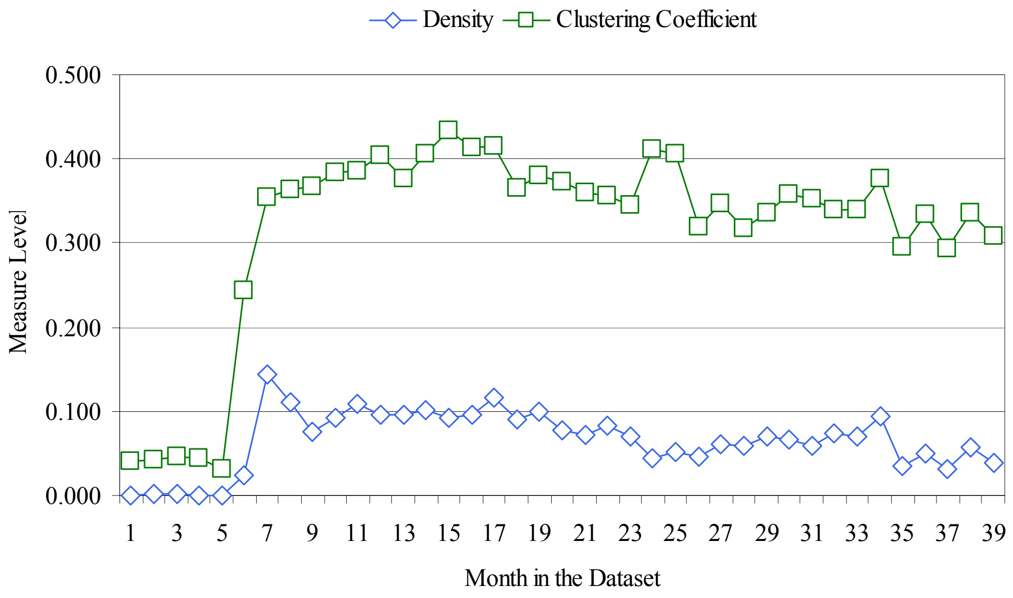

Figure 8: Evolution of Density and Clustering level of the C_R matrices (FCT method)

Figure 9 shows evolution of the density and clustering coefficient measures for the C_R matrices constructed based on the CGRAPH method. Although, the clustering coefficient values (avg=0.3979, min=0.0312, max=0.5402) are relatively similar to those shown in Figure 8. On the other hand, the CR matrices created using the CGRAPH methods are significantly more dense (avg=0.1509, min=0.0009, max=0.2408) than those created using the FCT method. In other words, CR matrices constructed with the CGRAPH method would suggest significantly levels of coordination requirements for the developers. Then, it is important to understand if the additional coordination needs are indeed necessary. The question is addressed in chapter 5.

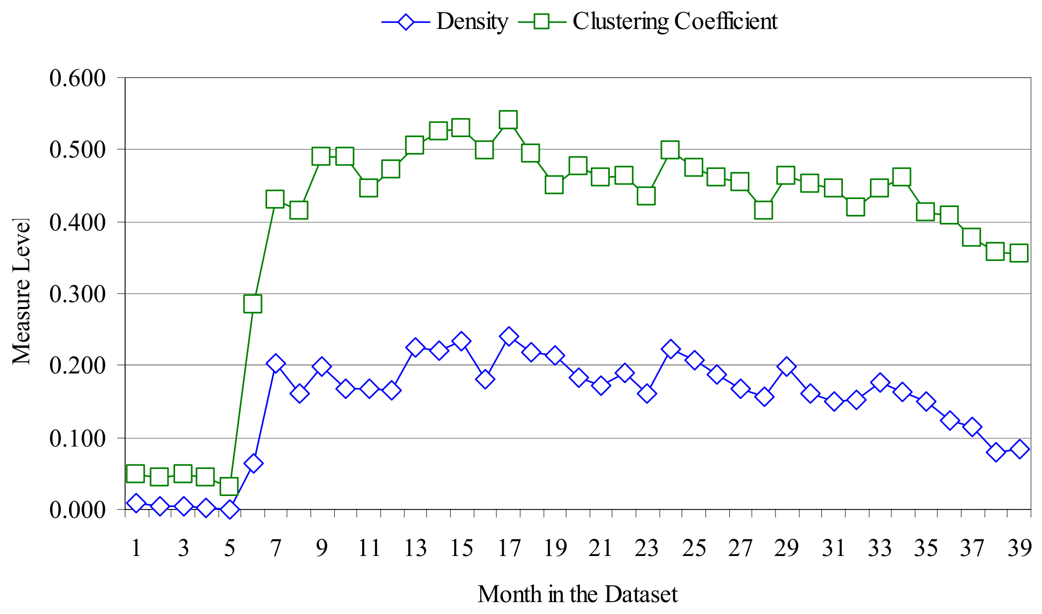

Figure 9: Evolution of Density and Clustering level of the C_R matrices (CGRAPH method)

Chapters 5 and 6 present two empirical studies that use the dependency identification techniques discussed in the previous paragraphs (FCT and CGRAPH) to examine the mismatch between coordination needs and coordination activities and their impact of two traditional outcome variables: development productivity and product quality.

CHAPTER 5: DEPENDENCIES, CONGRUENCE AND THEIR IMPACT ON DEVELOPMENT PRODUCTIVITY

Identifying work dependencies and determining the appropriate coordination mechanisms to address the dependencies is not a trivial problem. Coordination is a recurrent topic in the organizational theory literature and, as discussed in chapters 1 and 2, many stylized types of task dependencies and coordination mechanisms have been proposed over the past several decades. These perspectives are useful in the context of enduring structures. However, numerous types of work, for instance non-routine knowledge-intensive activities such as software development, are potentially full of fine-grain dependencies that might change on a daily or hourly basis. Conventional coordination mechanisms like standard operating procedures or routines would have very limited applicability in these dynamic contexts. Failure to identify the new needs for coordination and information exchange might hinder the organization's ability to adapt to changes in their competitive environment (Henderson & Clark, 1990). The study reported in this chapter represents the first step in the examination of how the gaps between coordination needs and actual coordination activity impact outcome variable, such as development productivity, in the context of software development activities.

Study I: Congruence and Development Productivity

Software development is populated with rapidly changing dependencies and this attribute of software development tasks is a potential source of coordination problems which impacts productivity. The analysis presented in this study focuses, first, in exploring the dynamism in the coordination requirements and, secondly, examining the impact that coordination activity congruent with coordination needs has on development performance.

Research Questions

When members of a team are physically collocated and coordination requirements within the team change, there are numerous ways for team members to identify the new needs and act on them such as group and status meetings and managerial intervention. However, social and communicational barriers pose important obstacles for coordination among individuals from different formal teams. Given these challenges, if the coordination requirements always involve the same set of developers, it is expected that over time individuals would develop common knowledge or a shared mental model that would reduce the possibility of coordination breakdowns (Espinosa, 2002). Unfortunately, rapidly changing coordination requirements would represent a more demanding environment. Therefore, it is also important to understand how the coordination requirements differ over time. This discussion leads to our first research question:

RQ 1: How stable are coordination requirements?

The organizational literature suggests that congruence is an important factor affecting task performance (Burton & Obel, 1998; Carley & Ren, 2001). For instance, mismatch between interdependent design tasks and coordination might have impact on the quality of airplane engines (Sosa et al, 2004). Moreover, in software engineering, coordination breakdowns can lead to longer development times (Espinosa, 2002; Herbsleb & Mockus, 2003) and higher number of defects and higher costs (Curtis et al, 1986). Then, higher levels of task performance associated with higher levels of congruence are expected, leading to the following research question:

RQ 2: Is higher congruence associated with better task performance?

Numerous factors such as the attributes of the task and individual-level characteristics drive communication and coordination patterns. As these factors evolve over time, it is crucial to understand the impact on the development of congruence, raising our third research question:

RQ 3: How do various types of congruence change over time?

Method

Data from Project A was used to examine the research questions addressed in this study. The unit of analysis is the modification request. A total of 2375 multi-team modification requests were identified. Those modification requests belonged to the first four releases of the product. Software development involves making a set of technical decisions that result in modifications to parts of the software. In order for the software to function correctly, the technical decisions made by the various developers must be compatible. Consequently, some type of coordination is required. Empirical research has shown that difficulties in communication and coordination breakdowns are recurring problems in software development (Curtis et al, 1988; Herbsleb & Mockus, 2003; Kraut & Streeter, 1995), particularly when the work items are geographically distributed (Herbsleb & Mockus, 2003) and the task involves more than one team (Curtis et al, 1988; Espinosa, 2002; Kraut & Streeter, 1995). For these reasons, the analysis focuses on the set of modification requests that involved more than one software development team.

Description of the Measures

The literature has identified a number of factors that affect development time and, consequently, the resolution of modification requests. Some of those factors are related to characteristics of the task such as the amount of code to be written and the priority of the task, whereas other factors capture relevant attributes of the individual developers and the teams that participate in the development task. In the following paragraphs, I first describe our dependent variable, resolution time of modification requests. Secondly, the procedures used to construct the measures of congruence are described. Finally, I describe a number of control measures that were also included in the statistical models.

Productivity Measure: The measure of task performance is *Resolution Time* which captures the time it took to resolve a particular modification request, and it accounts for all the time that the MR was assigned to developers. The modification requests reports contain records of when the MR was opened and resolved as well as every time the MR was assigned to a particular developer. Given this information, I can compute the amount of time that developers were actually working on the task.

Congruence Measures: The data for building the *Coordination Requirements* matrix was extracted from several data sources such as the modification request reports, the version control system as well as the software code itself. A modification request provides the “developer *i* modified file *j*” relationship that constitutes our *Task Assignment* matrix. Since, two different methods for identifying dependencies were used, FCT and CGRAPH, I constructed two different *Task Dependency* matrices. In the case of the FCT method, the cell c_{ij} of the *Task Dependency* matrix represents the number of times a particular pair of source code files changed together as part of the work associated with a modification request. As described in chapter 3, a moving window of 19 months was used to capture the relevant set of logical dependencies among the software modules. The resolution date of the modification request was paired with the end of the time window used to collect the task dependency information. In the case of the CGRAPH method, the cell c_{ij} of the *Task Dependency* matrix represents the number of data/function/method references from file *i* into file *j*. The data from the quarter associated with the resolution date of the modification request was used to collect the task dependency information. Then, using those *Task Assignments* and *Task Dependencies* matrices, the *Coordination Requirement* matrix is computed using equation 1.

In order to compute a measure of congruence, I also need to build the *Actual Coordination* matrix which represents the coordination activities that took place during the work associated with a modification request. These activities could take numerous forms and the communication and information exchanges could occurs over different means. Hence, four coordination paths were used to construct the *Actual Coordination* matrices. First, *Structural Congruence* captures the potential paths of communication and coordination that members of a formal team have through various mechanisms such as team meetings and other work-related activities. I built the actual coordination matrix where a coordination activity between engineers *i* and *j* exists if they belong to the same formal team. *Geographical congruence*, similarly to the case of organization structure, is built around the idea of potential paths of communication and coordination that exist when individuals work in the same physical location (Allen, 1997; Olson & Olson, 2000). Then, in terms of the matrix of coordination activities, engineers *i* and *j* have a linkage if they work in the same location. Higher levels of congruence would mean that the geographic location of people matches their coordination needs so that relatively little coordination is required across sites. *MR communication congruence* considers an exchange of technical information between engineers *i* and *j* only when both *i* and *j* explicitly commented in the modification request report. Multiple modification requests might refer to the same problem and later be marked as duplicates of a particular modification request. All duplicates of the focal MR were also used to capture the interactions among developers. Finally, *IRC communication congruence* was computed based on interaction between developers from the IRC logs. Three raters, blind to the research questions, examined the IRC logs corresponding to the period of time associated with each MR and established an interaction between engineers *i* and *j* if they made reference to the bug ID or to the task or problem represented by the MR in their conversations. In order to assess the reliability of the raters' work, 10% of the MRs where coded by all raters. Comparisons of the obtained networks showed that 98.2% of the networks had the same set of nodes and edges. All four *Actual Coordination* matrices were symmetric.

Control Measures: Past research has proposed several additional factors that impact development time (Espinosa, 2002; Herbsleb & Mockus, 2003; Kraut & Streeter, 1995). I collected a number of control variables that capture attributes of the task, the individuals and the teams associated with the development work. Several task-specific factors such as task dependency, priority and task re-assignments could have an inportant effect on development time. *Temporal Dependency* was measured as the number of modification requests that the focal MR depends on in order for the task to be performed. Management prioritized the activities of the developer by using a scale from 1 to 5 in the modification request report where level 5 as the highest priority and level 1 as the lowest priority. This rating constituted our measure of *priority* of the MR. *Task re-assignment* was measured as the number of times an MR was re-assigned to a different engineer or team. Re-assignment impacts resolution time because each new developer needs to build up contextual information about the task. In addition, MRs opened by customers could represent work items with higher importance consequently affecting the resolution time. A dummy variable was used to indicate if the MR is associated with the service request from a customer. *Multiple Locations* is a binary variable that indicates whether the all the developers that worked on a particular MR were in the same geographical location (a value of 0) or were distributed across the development labs (a value of 1). Finally, the *release* variable identifies the release of the product that the modification request is associated with. This variable could also be considered as a proxy for time to control for efficiencies that might develop over time and, consequently, affect the resolution time of the modification requests.

The amount of code written or changed is a proxy for the actual amount of development work done. The *change size* was computed as the number of files that were modified as part of the change for the focal MR. Prior research (Espinosa, 2002) has used lines of code changed as a measure of the size of the modification; however, a comparative analysis of both measures showed equivalent results in the statistical model used in this study. Therefore, the results presented in this chapter are based on the measure computed from the number of files modified. The change size measure was highly skewed so a log transformation was applied to satisfy the normality requirements of the regression model used in our analysis.

An experienced software engineer familiar with tools and programming languages can be substantially more productive than an inexperienced developer (Brooks, 1995; Curtis, 1981; Curtis et al, 1986). Furthermore, experience with the domain area and the technical characteristics of the application being developed help accelerate development time (Curtis et al, 1986). I used archival information as well as data from the software repositories to compute several individual level measures of experience. First, *programming experience* was computed as the average number of years of programming experience prior to joining the company of all the engineers involved in the modification request. *Tenure* was measured as the average number of months in the company of all the engineers that worked in the modification request at the time the work associated with the MR was completed. *Component experience* was computed as the average number of times that the engineers responsible for the modification request have worked on the same files affected by the focal modification request. This measure was also log-transformed to satisfy normality requirements. Finally, *Team load* is a measure of the average work load of the teams responsible for the components associated with the modification request. This control variable was computed as the ratio of the average number of modification requests in *open* or *assigned* state over the total number of engineers in the groups involved in the focal modification request during the period of time the MR was in *assigned* state.

Description of the Model and Preliminary Analysis

Past research has found that linear (Espinosa, 2002; Herbsleb et al, 2006) and hierarchical linear (Espinosa, 2002; Kraut & Streeter, 1995) models are appropriate techniques for examining the effects of different factors on development productivity. In this study, I examined the effect of the various congruence measures on task performance using the following linear regression model:

$$\begin{aligned} ResolutionTime = &\sum_i \beta_i * CongruenceMeasure_i + \\ &\sum_j \delta_j * ControlVariable_j + \varepsilon \end{aligned} \qquad \textbf{(Equation 3)}$$

An examination of descriptive statistics and Q-Q plot indicated that several of the variables (*Resolution Time*, *Chang Size* and *Component Experience*) were highly skewed to the left. The log transformation provided the best approximation to a normal distribution. Table 1 summarizes the descriptive statistics of the dependent and control variables included in our model. Table 2 summarizes the descriptive statistics of the congruence measures computed using the FCT method. Table 3 presents the descriptive statistics for the congruence measures computed using the CGRAPH method. The analysis of the pair-wise correlations amongst the variables in the model (Table 4) suggested no relevant collinearity problems. Only a small set of correlations were statistically significant but their levels did not exceed +/- 0.343.

Table 1: Descriptive Statistics for Dependent and Control Variables

	Mean	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
Resolution Time (log)	3.260	1.236	0	6.490	-0.809	3.127
Temporal Dependency	0.834	1.721	0	7	2.144	6.759
Priority	3.388	1.111	1	5	0.115	1.694
Re-assignment	1.457	1.599	0	6	0.481	1.605
Customer MR	0.483	0.499	0	1	0.067	1.004
Release	2.323	1.093	1	4	0.269	1.769
Change Size (log)	1.163	1.781	0	4.741	0.302	4.005
Team Load	9.104	2.938	1.016	58.800	-0.361	2.342
Multiple Locations	0.779	0.414	0	1	-1.346	2.814
Programming Exp.	4.429	3.654	2	22	1.074	4.462
Tenure	23.921	17.107	0	76	0.175	1.685
Component Exp. (log)	3.051	0.958	0	5.601	-0.015	2.145

Table 2: Descriptive Statistics for Congruence Measures (FCT method)

	Mean	SD	Min	Max	Skew	Kurtosis
Structural Cong.	0.663	0.217	0.156	0.995	-0.931	3.754
Geographical Cong.	0.684	0.237	0.142	0.993	-0.863	3.201
MR Cong.	0.567	0.283	0.070	0.982	-0.319	1.965
IRC Congr.	0.599	0.274	0.079	0.982	-0.506	2.233

Table 3: Descriptive Statistics for Congruence Measures (CGRAPH method)

	Mean	SD	Min	Max	Skew	Kurtosis
Structural Cong.	0.544	0.273	0.111	0.614	-0.322	1.849
Geographical Cong.	0.571	0.266	0.193	0.967	-0.062	2.048
MR Cong.	0.093	0.086	0.002	0.348	1.434	4.114
IRC Cong.	0.133	0.142	0.001	0.313	1.324	3.448

Table 4: Pair-wise Correlations
(N=2375, bold values are significant at $p < 0.05$).

		1	**2**	**3**	**4**	**5**	**6**
1	*Temporal Dependency*	-					
2	*Priority*	**0.341**	-				
3	*Re-assignment*	-0.013	0.029	-			
4	*Customer MR*	0.012	-0.031	**-0.224**	-		
5	*Release*	0.004	0.001	0.025	-0.019	-	
6	*Change Size*	0.113	**0.332**	0.031	-0.046	0.003	-
7	*Team Load*	-0.001	-0.029	**-0.329**	0.103	-0.008	-0.044
8	*Programming Exp.*	**0.314**	**0.343**	0.033	-0.021	-0.015	**0.218**
9	*Tenure*	**0.243**	0.023	0.009	0.001	-0.026	**-0.216**
10	*Component Exp.*	-0.043	-0.013	0.016	-0.001	-0.002	**-0.122**
11	*Multiple Locations*	**-0.160**	-0.013	-0.006	0.002	0.037	0.014
12	*Struct. Cong. (FCT)*	-0.030	0.022	-0.031	0.032	-0.015	0.049
13	*Geo. Cong. (FCT)*	-0.097	-0.035	0.008	-0.013	0.024	-0.008
14	*MR Cong. (FCT)*	0.007	-0.014	-0.003	-0.032	-0.013	-0.001
15	*IRC Cong. (FCT)*	-0.019	-0.006	0.079	-0.129	-0.016	-0.021
16	*Struct. Cong. (CGR)*	-0.024	-0.001	0.124	**-0.196**	0.035	0.055
17	*Geo. Cong. (CGR)*	0.004	-0.034	0.094	-0.064	0.002	-0.045
18	*MR Cong. (CGR)*	0.007	-0.014	-0.003	-0.032	-0.012	-0.001
19	*IRC Cong. (CGR)*	-0.063	0.010	0.058	-0.051	0.039	0.013

		7	**8**	**9**	**10**	**11**	**12**
7	*Team Load*	-					
8	*Programming Exp.*	-0.012	-				
9	*Tenure*	0.011	**0.266**	-			
10	*Component Exp.*	0.018	**0.161**	**0.245**	-		
11	*One Location*	0.010	0.012	-0.022	0.041	-	
12	*Struct. Cong. (FCT)*	0.031	-0.021	-0.052	-0.038	0.049	-
13	*Geo. Cong. (FCT)*	-0.009	-0.005	0.003	-0.003	0.087	**0.127**
14	*MR Cong. (FCT)*	-0.062	-0.004	-0.009	0.007	-0.040	0.033
15	*IRC Cong. (FCT)*	-0.044	-0.003	-0.022	-0.011	-0.003	0.028
16	*Struct. Cong.(CGR)*	-0.062	-0.021	-0.053	-0.003	0.059	0.041
17	*Geo. Cong. (CGR)*	-0.085	-0.004	-0.016	-0.010	0.072	0.015
18	*MR Cong.(CGR)*	-0.051	-0.014	-0.093	-0.039	-0.021	0.032
19	*IRC Cong.(CGR)*	-0.029	-0.008	0.002	0.001	-0.008	0.021

		13	**14**	**15**	**16**	**17**	**18**
13	*Geo. Cong. (FCT)*	-					
14	*MR Cong. (FCT)*	0.017	-				
15	*IRC Cong. (FCT)*	0.005	0.009	-			
16	*Struct. Cong.(CGR)*	0.009	0.009	0.027	-		
17	*Geo. Cong. (CGR)*	0.035	0.004	0.041	**0.188**	-	
18	*MR Cong.(CGR)*	0.032	0.039	0.001	0.021	0.044	-
19	*IRC Cong.(CGR)*	0.003	0.002	0.014	0.064	0.073	0.019

Results

In this section, the results of the various analyses performed to examine the three research questions addressed by this study are reported.

The Evolution of Coordination Requirements

I analyzed the evolution of the coordination requirements in order to address the first research question of this study. I seek to assess whether the needs to coordinate, in fact, change over time and how rapidly they change. I first focus our attention to two key aspects of the coordination requirements: the average change in an individual's coordination needs and the amount of coordination needed that crosses team boundaries. Figure 10 depicts the evolution of both factors on a monthly basis over the time period covered by the dataset. The average change in an individual's coordination needs (blue-diamond line in Figure 10) is computed by comparing the *Coordination Requirements* matrices (constructed using the FCT method) from month *t* against month *t-1* and averaging the amount of change in the coordination needs across all the individuals. As an example, a 10% value of change in the coordination needs in month *t* means that 10% of the coordination requirements of any particular developer did not exist in the previous month (*t-1*). The amount of coordination requirements that involve developers from other formal organizational groups is represented by the green-square line in Figure 10. The values are computed by identifying those coordination requirements that cross the boundaries of an individual's team and averaging those specific coordination needs. Figure 10 shows significant volatility in the coordination requirements over time. There are several instances where the level of change in the coordination requirements is above 30%. A similar pattern also occurs for the amount of coordination needs that cross the team boundary.

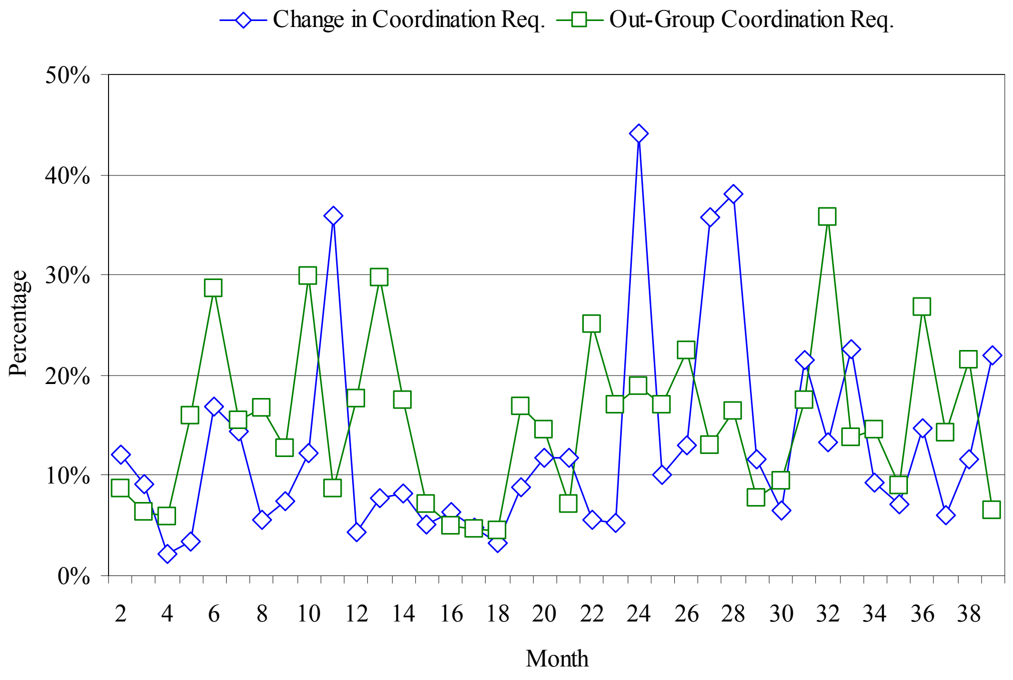

Figure 10: The Evolution of Coordination Requirements on a Monthly Basis

The variability of the coordination requirements using the CGRAPH method was also examined. I found that the coordination needs tend to be quite stable which is not a surprising finding given the very low variability of the information contained in the T_D matrix computed using the CGRAPH method (see Figure 6 in Chapter 4).

A second factor that could affect the variability of the coordination needs is the technical properties or organizational factors of project A. In order to assess if these patterns of coordination needs are found in other projects, I used data from three different projects of the Apache Software Foundation, described in chapter 3 as project B. Figure

11 shows the average change in an individual's coordination needs for each of the three Apache projects: Ants, Tomcat and Struts. I did not examine the change in the amount of coordination needs that cross the team boundary because these open source projects do not have a formalized organizational structure as traditional closed source project typically have. In this case, the observed variability in the coordination requirement is significantly lower relative to the magnitudes reported in figure 10 for the closed source project.

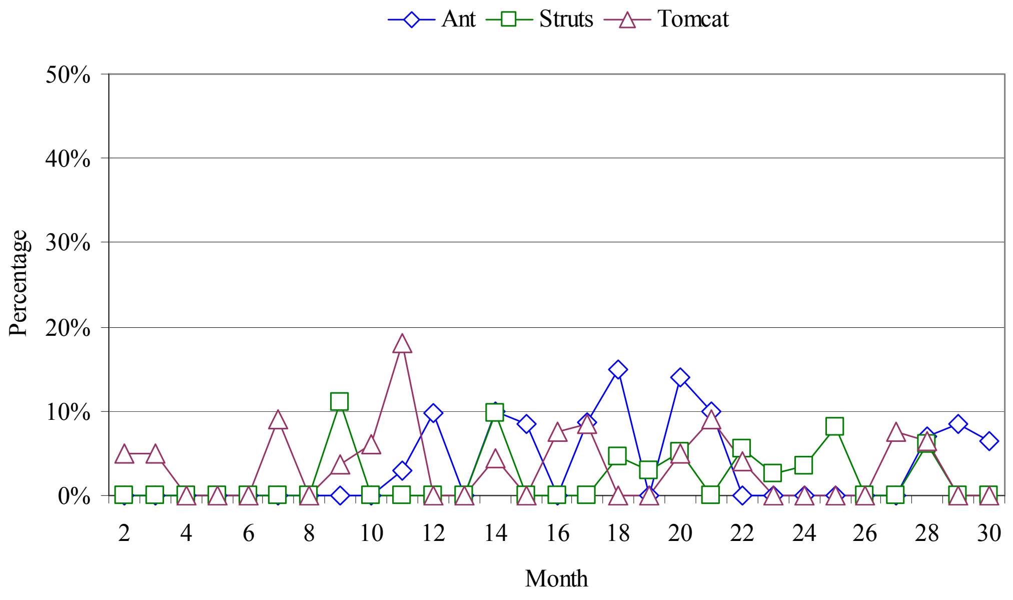

Figure 11: The Evolution of Coordination Requirements in Open Source Projects

There are several characteristics of the open source projects that could explain the difference in the volatility of coordination needs between the closed source project and the Apache Foundation projects. First, there were 24 time periods where the rate of change was zero, in 6 of those 24 months there were no commits into the version control system and in 8 other months, the commits were done by only 2 individuals. On the other hand, in the closed source project, there were a minimum of 587 commits in any month contributed by a minimum of 71 developers. This exemplifies the significant differences between the projects in terms to size, the amount of source changed and the overall amount of development activity during the period of time covered by the data.

Secondly, identifying modification requests in open source project such as Ant, Tomcat and Struct, is quite a complex endeavor because the nature of the procedures used by developers. Open source projects, typically, do not associate the information in defect tracking tools (e.g. Bugzilla) with the data in the version control systems (e.g. CVS). Researchers attempt to construct synthesized modification request by grouping a collection of changes to source code files when those changes are made within a particular period of time, for instance within a few minutes (Zimmermann & Weibgerber, 2004). Unfortunately, this type of approach has several limitations which reduce the reliability of the data. First, the approach assumes that a modification request is resolved by just one bundle of changes the software system. Secondly, the approach also assumes that only one developer is involved in resolving a particular modification request. I have observed several projects (such as Projects A, C and D) where that assumption held only for a fraction of the modification requests.

Albeit the differences in the characteristics of the projects and the limitations of the data, I think that all four projects show that the coordination requirements shift quite often. As it was argued in chapter 1, traditional coordination mechanisms do not provide an appropriate framework for handling rapidly changing work dependencies. Then, understanding how the actual coordination activities carried out by the developer match those and what are the implications in terms of development productivity becomes an important research question.

The Impact of Congruence on Resolution Time of MRs

I performed several linear regression analyses to assess the effect of the congruence measures on resolution time. The results are presented in Table 5. Models I is a baseline regression considering only the control factors. Models II introduces the measures of congruence in the analysis computed using the FCT method. Model IV introduces the measures of congruence in the analysis computed using the CGRAPH method. Finally, models III and V also include several interaction factors to assess whether the role of congruence changes across the different releases of the product and when the groups involved in a particular MR are geographically distributed. The measures of structural and geographical congruence could be affected by personnel turnover and mobility across teams. In order to assess whether these factors contributed to the results, I examined archival data collected from the company and I determined a yearly turnover rate of only 3% and an inter-group mobility rate of less than 1%. The modification requests that involved individuals that left the company or changed group membership were eliminated from the analysis. However, an analysis including those modification requests showed results consistent with those reported in tables 5 and 6.

Model I reports results consistent with previous empirical work in software engineering. Factors such as the size of the modification, familiarity with the software components, and general programming experience are significant elements that affect resolution time of MRs (Espinosa, 2002; Herbsleb & Mockus, 2003). Task-specific characteristics such as temporal dependencies with other modification requests and the priority of the task increased development time. As it has been reported in previous research (Espinosa, 2002; Herbsleb & Mockus, 2003), the results also show that when developers are geographically distributed, the amount of time required to resolve modification requests increases.

The results indicated that time, captured by the variable *Release*, had no statistical effect. Since the *Release* measure is in fact a categorical variable, I also examined its impact using two dichotomous variables to represent the four possible values. The results were identical to defining *Release* as an integer from 1 to 4 to represent the four releases of the product.

Table 5: Results from OLS Regression of Effects on Resolution Time (FCT method)

	Model I	Model II	Model III
(Intercept)	4.814**	4.626**	4.485**
Temporal Dependency	0.592**	0.591**	0.591**
Priority	-0.401**	-0.404**	-0.404**
Re-assignment	0.011	0.013	0.011
Customer MR	0.091	0.098	0.091
Release	-0.018	-0.018	-0.031
Change Size (log)	0.306**	0.311**	0.310**
Team Load	-0.006	-0.006	-0.005
Multiple Locations	0.128**	0.131**	0.128**
Programming Experience	-0.166**	-0.166**	-0.166**
Tenure	-0.002+	-0.002+	-0.002
Component Experience (log)	-0.065**	-0.065**	-0.066**
Structural Congruence (FCT)		-0.137*	-0.184*
Geographical Congruence (FCT)		-0.014*	-0.041*
MR Congruence (FCT)		-0.057*	-0.051*
IRC Congruence (FCT)		-0.066*	-0.205*
Release X Structural Congruence (FCT)			0.020
Release X Geographical Congruence (FCT)			-0.027
Release X MR Congruence (FCT)			-0.041
Release X IRC Congruence (FCT)			-0.065*
Multiple Locations X MR Congruence (FCT)			0.133
Multiple Locations X IRC Congruence (FCT)			-0.274*
N	2375	2375	2375
Adjusted R^2	0.718	0.819	0.831

($^{+}$ $p < 0.10$, * $p < 0.05$, ** $p < 0.01$)

Table 6: Results from OLS Regression of Effects on Resolution Time (CGRAPH method)

	Model I	Model IV	Model V
(Intercept)	4.814**	4.876**	4.976**
Temporal Dependency	0.592**	0.592**	0.591**
Priority	-0.401**	-0.402**	-0.401**
Re-assignment	0.011	0.034	0.037
Customer MR	0.091	0.188	0.183
Release	-0.018	-0.016	-0.051
Change Size (log)	0.306**	0.306**	0.305**
Team Load	-0.006	-0.005	-0.006
Multiple Locations	0.128**	0.125**	0.176**
Programming Experience	-0.166**	-0.167**	-0.166**
Tenure	-0.002+	-0.003+	-0.001
Component Experience (log)	-0.065**	-0.064**	-0.065**
Structural Congruence (CGRAPH)		-0.205+	-0.231+
Geographical Congruence (CGRAPH)		-0.113*	-0.031*
MR Congruence (CGRAPH)		0.412	0.480
IRC Congruence (CGRAPH)		-0.002	0.019
Release X Structural Congruence (CGRAPH)			0.218
Release X Geographical Congruence (CGRAPH)			-0.002
Release X MR Congruence (CGRAPH)			-0.035
Release X IRC Congruence (CGRAPH)			0.131
Multiple Locations X MR Congruence (CGRAPH)			0.044
Multiple Locations X IRC Congruence (CGRAPH)			-0.424
N	2375	2375	2375
Adjusted R^2	0.718	0.731	0.722

($^{+}$ $p < 0.10$, * $p < 0.05$, ** $p < 0.01$)

Model II in table 5 shows statistically significant effects on all the congruence measures computed using the FCT method. The estimated coefficients of the congruence measures have negative values which are associated with a reduction in resolution time. The results highlight the important role of congruence on task performance as well as the complementary nature of all communication paths. Structural congruence is associated with shorter development times suggesting that when coordination requirements are contained within a formal team and appropriate communication paths exists, task performance increases. Geographical congruence had a positive effect on resolution time, consistent with past research that argued distance has detrimental effects on communication (see Herbsleb & Mockus, 2003 and Olson & Olson, 2000 for reviews). Communication congruence based on the interactions amongst engineers through the MR reports as well as IRC were also statistically significant suggesting the usefulness of these tools in facilitating coordination among individuals that belong to different teams and could potentially be geographically distributed. Model III includes several interaction terms. The results showed statistical significance only in the Release X IRC congruence and Multiple Locations X IRC congruence interactions. The negative coefficients in both interactions suggest that in later releases or when developers are geographically distributed the impact of IRC congruence on resolution time is higher above and beyond the direct effect.

Model IV in table 6 shows the results obtained when the congruence measures are computed using the CGRAPH method. In this case, only geographical congruence is statistically significant and its coefficient is negative indicating a reduction in the resolution time as congruence increases. Structural congruence was marginally significant. These results support the argument that the two dependency identification methods, FCT and CGRAPH, are capturing different sets of technical dependencies that impact the development tasks differently. From an analytical point of view, the difference between the results from model IV (table 6) and model II (table 5) could stem from higher levels of density in the C_R matrices when using the CGRAPH method as discussed in the "preliminary analysis" section. Higher levels of density in the C_R matrices imply higher numbers of coordination requests that have to be matched by the actual coordination matrices (C_A). In the cases of structural and geographical congruence the resulting C_A matrices would tend to be denser than those in the case of MR and IRC communication because I assumed coordination activity amongst all members of team or a location, respectively. Then, C_A matrices for structural and geographical congruence would provide a "better fit" to the denser C_R matrices. This argument is supported by the descriptive statistics from table 3 that indicate that the range of values for MR and IRC congruence is significantly smaller than those for structural and geographical congruence. Finally, Model V in table 6 shows that interaction terms were not statistically significant when considering congruence measures computed using the CGRAPH method.

The Evolution of Congruence over Time

The previous section showed that when communication amongst individuals matches the communication requirements imposed by the task dependencies, task performance is improved. The next step is to explore the evolution of the measures of congruence over time. In this analysis, I used the FCT method to compute the four different measures of congruence because these measures had a statistically significant effect in the empirical analysis reported in the previous section. The communication networks were built on a weekly basis. Congruence was computed comparing the *Coordination Requirements* matrix from week t_n to the *Actual Coordination* matrix from week t_{n-1}, because I assumed that developers would discuss a particular problem before making the actual changes in the source code. I also computed the congruence measures using week t_n for both required and actual coordination, and the trends remained the same. The communication network based on IRC or modification requests represents an aggregate measure across all MRs resolved in a particular week. One difficulty when doing a longitudinal analysis of a software project is the changing nature of the tasks. For instance, in the first release, an important amount of feature development activity took place during the period of analysis. By the third and fourth releases, the modification requests were mostly related to defect resolution. Therefore, I also explored the relationship between the characteristics of the task, i.e. feature development or defect resolution, and the evolution of the congruence measures.

The analysis showed that the different measures of congruence varied significantly across releases. Figure 12 shows the average level of each measure of congruence across the different releases. In the first release, structural and geographical congruence dominate while communication congruence based on MRs or IRC are almost absent. In later releases, structural congruence decreases significantly, particularly in the third and forth releases. This result is consistent with the results on the volatility of coordination requirements discussed earlier in this chapter, suggesting that the dependencies among the various components of the software system are changing over time and the work requires the contribution of individuals from different teams. The decline in structural congruence could also be interpreted as a deterioration of the homomorphic relationship between product and work structures posited by the modularity theoretical argument (Baldwin & Clark, 2000; Conway, 1968; Parnas, 1972). The measures of communication congruence based on MR and IRC increase in release 2 and they remain high during the last two releases. The increase in communication congruence coincides with the gradual decrease of structural congruence. A possible interpretation of this result is that developers are learning to substitute the lack of formal communication paths with interactions through other means such as IRC and MR reports.

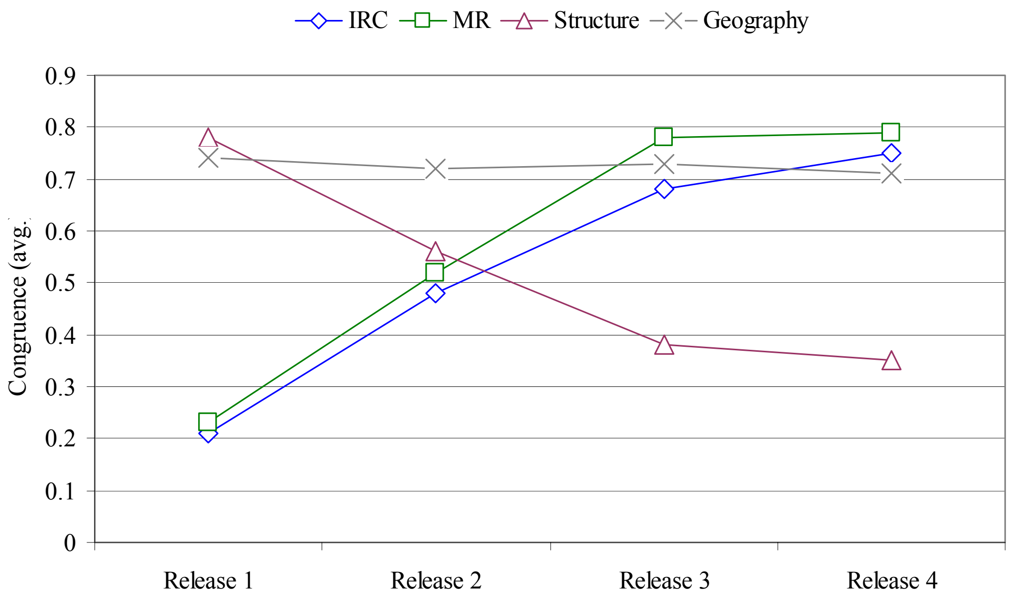

Figure 12: Evolution of the Congruence Measures across Releases

I also examined the evolution of congruence from a statistical point of view using a repeated measures type of analysis. Congruence was considered as the dependent variable and I considered the main effect of time, type of task and type of congruence measure as well as the interaction terms. Type of task refers to whether the modification request refers to a feature development or a defect task. Table 7 shows a significant main effect of time on congruence as well as the type of congruence. Moreover, the interaction of time and type of congruence is significant suggesting that the various measures are changing over time in different ways as shown in Figure 12. Type of task has a main positive effect on congruence which is higher for feature development tasks. However, the effect of type of task remains the same over time as suggested by the lack of significant in the "*Time-Type of Task*" interaction effect.

Table 7: Effect of Time on Congruence.

		F	**p**
Main Effects	*Time*	107.028	<**0.001**
	Type of Congruence	112.208	<**0.001**
	Type of Task	8.465	**0.004**
Interactions	*Time * Type of Congruence*	116.051	<**0.001**
	*Time * Type of Task*	0.387	0.742

Appropriate communication and coordination is an integral part of the software development process (Herbsleb & Mockus, 2003; Kraut & Streeter, 1995). The significant changes in communication patterns shown by Figure 12 raise an interesting question: are all developers able to identify the changes in coordination requirements and adapt their communication paths accordingly? Mockus and colleagues (2002) reported that in open source and commercial projects most of the modifications to the software are made by a small number of developers. These findings provide a useful framework to identify the most productive developers, in order to compare their coordination behaviors with the less productive developers.

I computed the contributions of the developers in project A and I found that 50% of the modifications made to the software system were done by only 18 (15%) developers (see Figure 13). I then separated the developers and their interactions into two groups and repeated the analysis reported above. Figure 14 shows the evolution of congruence for the top 18 contributors across releases. The general patterns are similar to the overall results shown in Figure 12. Structural congruence decays over time while MR and IRC communication congruence increase considerably in the last two releases.

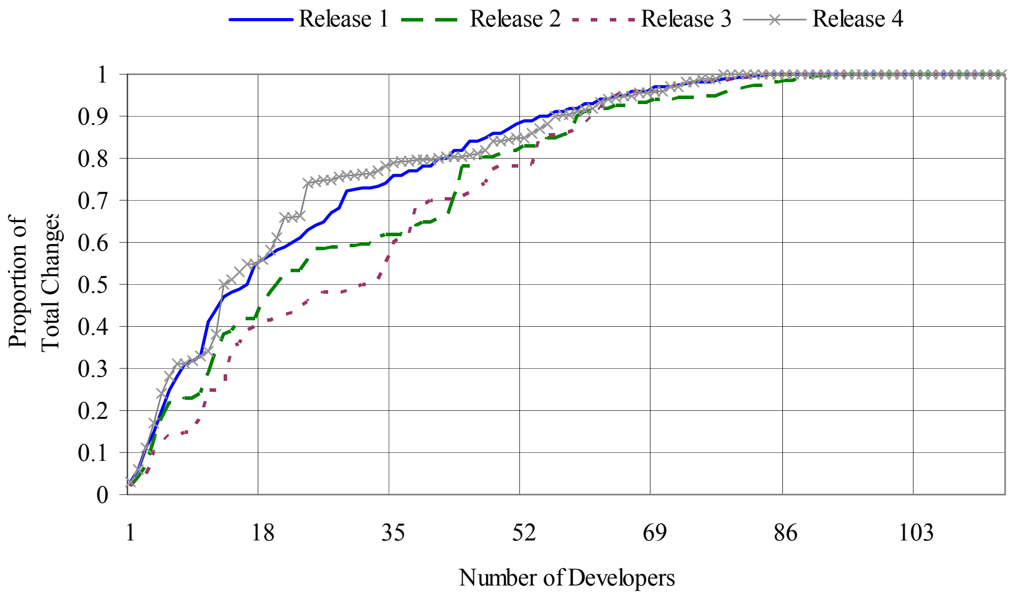

Figure 13: Proportion of Changes per Developer per Release

On the other hand, Figure 15 depicts a very different result for the rest of the developers. Structural congruence decreases over time but not as drastically as in the case of the top performers. Moreover, these developers do not seem to use the computer-mediated communication means to interact with the right set of people. Consequently, they never achieve high levels of congruence in the IRC and MR congruence measures.

The software engineering literature suggests that top developers typically have an order of magnitude better performance than average developers and the sources of that disparity are usually attributed to differences in experience and cognitive ability (Curtis, 1981; Curtis et al, 1986). The results reported in this chapter did not provide evidence that differences in experience and familiarity with the system or attributes of the tasks were significant sources of difference in performance.

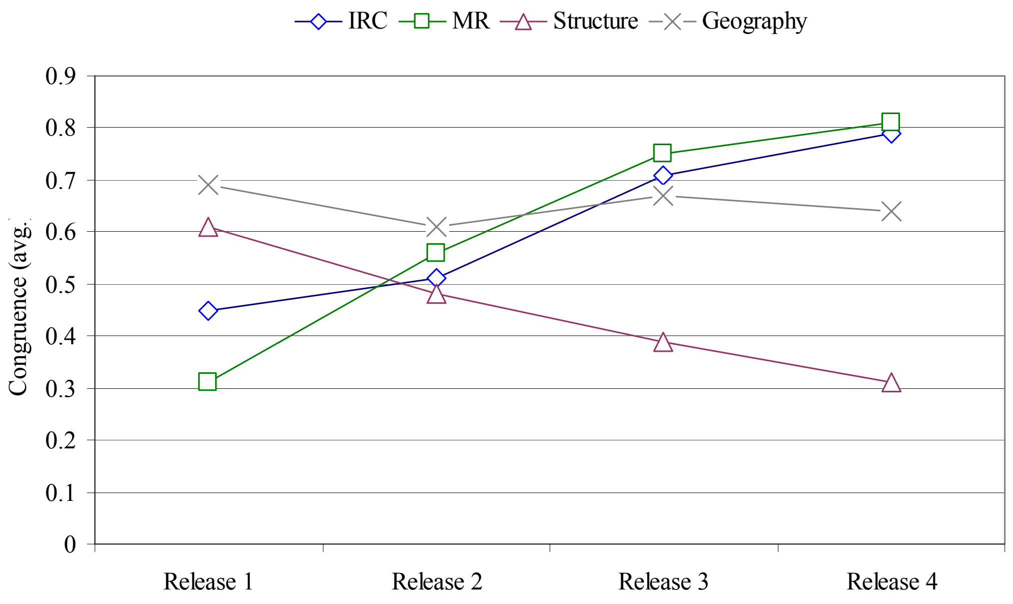

Figure 14: Congruence Measures across Releases based on Top Contributors Interactions

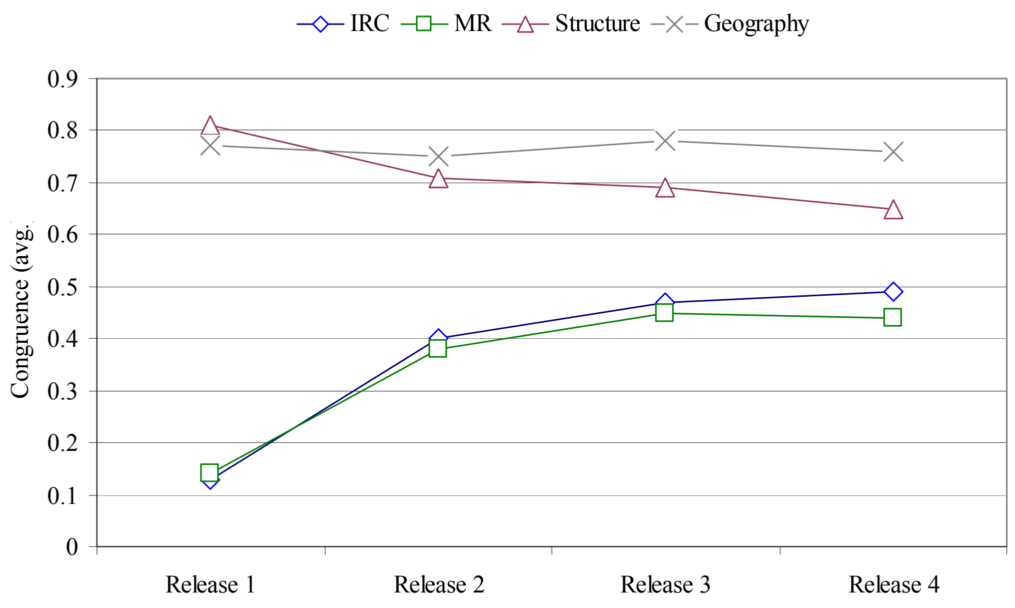

Figure 15: Congruence Measures across Releases for the Rest of the Developers

Table 7 shows the results of comparing the other developers against the top performers. The two groups of individuals do not differ in terms of domain experience, their level of education and their tenure in the company. The disparity in programming experience is marginally significant, suggesting that the top performers might have slightly deeper programming experience than the rest of the developers. I also compared some of the attributes of the modifications requests that the two groups of developers worked on such as the average size of the changes made to the software and the average number of lines added to and removed from the software. The comparison of the average size of the modification request was marginally significant, suggesting that top performers tended to work on slightly larger changes to the software. However, there was no statistically significant difference in terms of lines of code added or deleted.

Table 8: Differences between developers' population

	t	**p**
Programming Experience	-1.85	0.073
Domain Experience	-0.59	0.556
Graduate Education	1.03	0.311
Tenure in the company	1.21	0.239
Avg. Size of Changes	-1.79	0.072
Avg. Lines Added	-1.51	0.148
Avg. Lines Deleted	0.95	0.341

The analysis and results reported in this chapter do not present any evidence of causality between patterns of communication and developer's performance. The results suggest that the traditional perspective in software engineering relating only cognitive ability and experience to contributions (Curtis, 1981) may not capture all of the important attributes of top-performing developers, since their performance seems to have a substantial social component. Chapter 7 explores in more detail the interesting questions raised by figures 14 and 15 regarding the relationship between patterns of coordination and individual-level development performance.

Discussion

This study evaluated a measure of coordination that extends traditional conceptualizations of coordination by taking a fine-grain level of analysis to better examine the mismatches between dependencies and coordination activities. Those gaps could have major implications for the productivity and the quality of the output of product development organizations (Curtis et al, 1986; Espinosa, 2002; Herbsleb & Mockus, 2003; Sosa et al, 2004) and for non-routine intellectual work more generally. The empirical results suggest that the technique described in chapter 2 provides a useful framework to examine how coordination needs that are not satisfied impact software development productivity. When the developers coordinate their task with the relevant set of workers, productivity increases. I also addressed the dynamic nature of dependencies that exist in complex tasks such as software development. Individuals have difficulties identifying task interdependencies that are not obvious or explicit (Sosa et al, 2004) and the developers' ability to recognize dependencies diminish as coordination requirements change over time (Henderson & Clark, 1990). For these reasons, volatility in the coordination requirements represents a major hurdle for work groups and, particularly, for those that are geographically distributed. Collaborative tools could play an important role in reducing the gap between recognized and actual interdependencies. It would be highly desirable for future tools to be able to assess the characteristics of the task and assist the users in identifying and dealing with dependencies unknown a priori or that emerged as a consequence of the evolving characteristics of tasks. The congruence measure provides a framework for those future tools.

The results showed the product structure-task structure relationship is not as simple as theorized. Modularization techniques in software development only consider one type of technical dependencies, syntactic relationships (Garcia et al, 2007). That limitation manifested clearly in the results. The empirical evaluation of the congruence framework showed the importance of understanding the dynamic nature of software development. Identifying the “right” set of technical dependencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution of modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively stable view of product dependencies that is not representative of the dynamism of software development activities. On the other hand, logical dependencies provide a more accurate representation of the most relevant technical dependencies in software development projects.

CHAPTER 6: DEPENDENCIES, CONGRUENCE AND THEIR IMPACT ON SOFTWARE QUALITY

Quality is a fundamental topic in software engineering. The multidimensional nature of the concept has led prolific research across many areas in the software engineering literature. For instance, an extensive literature in software reliability and related areas has focused on developing pragmatic failure prediction models as well as estimation of the reliability of a system in terms of time to failure (Fenton & Neil, 1999). The work on software process is another area that had examined how multiple factors relate to software quality (e.g. Paulk et al, 1995; Pressman, 2004). A growing body of empirical work in software dependencies has examined the relation between the structure of software systems and their proneness to failure. Early research explored approaches to measuring reference coupling among components or modules and it showed a positive relationship between high levels of coupling and failure proneness of a software system (see Chidamber & Kemerer, 1994 and Arisholm et al, 2004 for reviews). Those findings apply to systems built based on a structured design approach (Selby & Basili, 1991) as well as object-oriented systems (Briand et al, 2000). The work on software dependencies has focused on syntactic relationships between modules, ignoring implicit or logical dependencies which could potentially be more relevant in the context of failure proneness.

The study described in this chapter examines the relationship of failure proneness and various representations of product dependencies, syntactic and logical relationships.

The study also examines the impact of work dependencies and patterns of coordination on failure proneness, factors that have been neglected by the literature.

Study II: The Structure of Dependencies, Congruence and Product Quality

Customer reported software faults are, arguably, caused by violation of dependencies that are not recognized by the developers implementing a software system. Those dependencies could stem from various sources such as technical properties of the system under development and how the development work is organized. The software engineering literature suggests several types of technical dependencies. One form of software dependencies are syntactic relationships among modules of a system that are reflected in the code by the definition and use of functions, methods, variables and other programming language constructs. This line of work found that higher levels of coupling are related to higher levels of failure proneness of a software system. However, syntactic dependencies are only one approach for representing the structure of a software system. In more recent work in the software evolution literature, Gall and colleagues (1998) examined the evolution of changes to modules to identify logical dependencies. The approach attempts to uncover dependencies among modules that are not explicitly identified by traditional syntactic approaches. Unfortunately, our understanding of the relationship between the structure of logical dependencies and failure proneness of a system is very limited. Yu (2006) reported a positive correlation between logical and syntactic dependencies which would suggest that higher numbers of logical dependencies would increase the likelihood of failure. However, those results are based on only one system and generalizing of the relationship between syntactic and logical dependencies is difficult. Moreover, the results presented in chapters 4 and 5 suggest the two types of product dependencies capture different set of relationships between components of the system. Hence, further study of this relationship is required in order to understand the implications of different types of product dependencies on the failure proneness of a software system.

Human and organizational factors may also affect the quality of a software system. The level of interdependency between tasks tends to drive communication and coordination among workers (Galbraith, 1973; von Hippel, 1990). However, recent studies of coordination in software development suggest that the identification and management of technical dependencies is a challenge in software development organizations, particularly, when those dependencies are semantic rather than syntactic (Bass et al, 2006; Cataldo et al, 2007; de Souza, 2005; Grinter et al, 1999). Then, appropriate levels of communication and coordination may not occur, potentially decreasing the quality of a system (Curtis et al, 1988; Herbsleb et al, 2006). Consequently, it is important to understand how work dependencies and the coordination behavior of developers impact the failure proneness of a system.

The primary contribution of this study is the examination of the impact that syntactic, logical and work dependencies have, simultaneously, on the failure proneness of a software system. I also focus on enhancing external validity by replicating the study on two distinct projects from two unrelated companies. First, I examine how syntactic and logical dependencies relate to a software system's failure proneness. Secondly, I incorporate in the analysis of failure proneness the role of work development. Thirdly, the developers' ability to coordinate their work congruently with regards to the coordination needs is considered. In sum, I examine how work-related factors affect the quality of software system above and beyond the technical dependencies among the various parts of that software system.

Research Questions

The traditional view of software dependency, syntactic dependencies, had its origins in compiler optimizations and they focus on control and dataflow relationships (Horwitz et al, 1990). This approach extracts relational information between specific units of analysis such as statements, functions or methods, as well as modules, typically, from the source code of a system or from intermediate representations of software code such as bytecodes or abstract syntax trees. These relationships can represent either a data-related dependency (e.g. a particular data structure modified by a function and used in another function) or a functional dependency (e.g. method A calls method B). The pioneering work by Basili and colleagues (Hutchins & Basili, 1985; Selby & Basili, 1991) represents the first attempt to use of this type of data in the context of failure proneness of a system. Building on the concepts of coupling and cohesion proposed by Stevens, Myers and Constantine (1974), Hutchins and Basili (1985) presented metrics to assess the structure of a system in terms of data and functional relationships which were called bindings. The authors used clustering methods to evaluate the modularization of a particular system. Selby and Basili (1991) used the data binding measure to relate system structure to errors and failures in a software system. Using a comparison of means approach, the authors argued that routines and subsystems with lower coupling were less likely to exhibit defects than those with higher levels of coupling. Similar results have been reported in object-oriented systems. Chidamber and Kemerer (1994) proposed a set of measures that captures different aspects of the system of relationships between classes in an object-oriented design. Briand and colleagues (2000) found that the measures of coupling proposed by Chidamber and Kemerer were positively associated with failure proneness of classes of objects.

A second, and more recent, view of dependency has been developed in the software evolution literature. This approach focuses on deducing dependencies between modules of a system that are changed together as part of the software development effort. Gall and colleagues (1998) called this type of relationships "logical" dependencies. They differ from traditional syntactic dependencies because they are able to identify indirect or semantic relationships between modules that are not explicitly deducible from the programming language constructs (Gall et al, 1998). Remote procedure calls (RPC), infrastructure code (e.g. memory management, basic libraries) or cross-cutting concerns like logging and security represents cases where logical dependencies provide more valuable information than syntactic dependencies. For instance, in the case of RPCs, the syntactic dependency approach would provide a long path of connections because a call-graph would identify the sequence of functional relationships from the module invoking the RPC through the RPC stubs all the way to the RPC server module. On the other hand, logical dependencies would show a direct dependency between the module invoking the RPC and the server module or the RPC specification if those pieces of the system changed together. In the case of cross-cutting concerns, the information provided by the syntactic dependencies approach would highlight highly coupled modules (the ones that implementing the logging or security functionality) that tend to be very stable, and consequently, very unlikely to be prone to failures although the high coupling would suggest otherwise. The logical dependency approach eliminates these problems because the likelihood of modules that implement cross-cutting concern changing together with other modules is very low, hence, a logical dependency would not be established.

Unlike the case of syntactic dependencies, limited work has focused on the relationship between logical dependencies and failure proneness of a system. Yu (2006) reported positive correlations between logical and syntactic dependencies in the Linux operating system. Nagappan & Ball's (2007) study found that logical coupling metrics are correlated with post-release failure proneness of programs. However, these studies have important limitations. First, the studies examined only one system, hence, there are threats to external validity. Secondly, these studies did not examine the impact of the structure of the logical dependencies. Thirdly, Nagappan and Ball (2007) computed the metrics using a coarse-grain approach at the level of program bundles, called areas, and the measures were all highly correlated which did not allow the authors to assess the actual impact on failure proneness of each metric relative to other factors that might also contribute.

The work on syntactic dependencies suggests that higher levels of coupling between modules of a systems, the higher the level of failure proneness of a systems. The limitations of the current work on logical dependencies do not allow us to reach the same conclusion. More importantly, the majority of the research on software dependencies tends to examine correlation between variables of interest, consequently, such analysis do not explore the effects of the various factors simultaneously. These gaps in the literature lead to the following research question addressed by this study:

RQ 1: How does the structure of dependencies, syntactic and logical, affects the failure proneness of a system?

The literature on failure proneness has focused on the role of technical properties of a software system neglecting the impact of human and organizational factors on the quality of a software system. The work on coordination in software development suggests that the identification and the management of work dependencies is a challenge in software development organizations (Grinter et al, 1999; Herbsleb et al, 2000; Herbsleb & Mockus, 2003). Unfortunately, modularization is not a sufficient representation of work dependencies in software development for several reasons. First, recent empirical evidence indicates that the relationship between product structure and task structure is not as simple as previously assumed (Cataldo et al, 2006). Secondly, software modularization techniques only consider one type of product dependency, syntactic relationships (Garcia et al, 2007). Thirdly, promoting minimal communication between teams responsible for interdependent modules is problematic because it significantly increases the likelihood of occurrence of integration problems (de Souza et al, 2004; Grinter et al, 1999). Herbsleb and colleagues (2006) theorized that the irreducible interdependence among software development tasks can be thought of as a distributed constrain satisfaction problem (DSCP) where coordination is a solution to the DSCP. Within that framework, the authors argued that the patterns of task interdependencies among the developers as well as the density of the dependencies in the constraint landscape are important factors affecting coordination success, consequently, affecting the quality of a software system and the productivity of the software development organization.

In sum, the quality of a software system depends on technical properties of the system such as the structure of dependencies between the modules or relevant parts of the system as well as the ability of the developers to identify and manage work dependencies, which leads to the following research question:

RQ 2: How does the structure of work dependencies affects the failure proneness of a system?

Finally, mismatches between coordination requirements and coordination behavior might have negative implications on the quality of the product (Sosa et al, 2004). Moreover, in software engineering, coordination breakdowns can lead to higher number of defects and higher costs (Curtis et al, 1988; Herbsleb et al, 2006). Then, if developers coordinate their work effort in a congruent way given a particular set of work dependencies, lower levels of failure proneness associated with higher levels of congruence are expected, leading to the following research question:

RQ 3: Is a higher level of congruence associated with lower levels of failure proneness?

Method

I examined the research questions using data from 8,257 modification requests from project A and 7000 modification requests from project C. In the rest of this section, I first describe the various measures followed by a description of the statistical model used in the analysis.

Description of the Data and Measures

In order to study the research questions outlined in the previous section, several sources of data from projects A and C, such as source code, version control systems and defect tracking data, were used. The following paragraphs describe the measures as well as the statistical models used in the analysis. Tables 9 and 10 present the descriptive statistics of the measures used in this study.

Measuring Failure: The dependent variable, *File Buggyness*, is a binary measure indicating whether a file has been modified as part of the resolving a field defect. Therefore, the unit of analysis is the source code file. In the datasets, there were four releases available to customers in project A and six releases were available to customers in project C. Using the modification requests from projects A and C, the dataset of source code files was constructed in the following way. First, the dataset included all the files that were modified as part of the development effort or as part of resolving a defect in a particular release. For each one of those files, I determined if they were associated with a field defect in any of the releases of the product covered by the data. Secondly, I included all files that were associated with field defects that did not change during the development of a release under study. The following logistic regression model was used to assess the effect of the various independent factors:

$$\begin{aligned} FileBuggyness = & \sum_i \beta_i * SyntacticDependenciesMeasure_i + \\ & \sum_j \chi_j * LogicalDependenciesMeasure_j + \\ & \sum_n \delta_n * WorkDependenciesMeasure_n + \\ & \sum_h \lambda_h * CongruenceMeasure_h + \\ & \sum_k \phi_k * AdditionalMeasure_k + \varepsilon \end{aligned} \qquad \textbf{(Equation 4)}$$

The independent variables indicated in the model are described in the following paragraphs.

Syntactic Dependencies: Syntactic dependency information was collected using the C-REX tool (Hassan and Holt, 2004) to identify programming language tokens and references in each entity of each source code file. This analysis was performed over the entire source code of the two systems at the end of the 3rd month of each calendar quarter. Using the resulting data, I computed dependencies between source code files by identifying data, function and method references that cross the boundary of each source code file. If we think in terms of a matrix of source code files, each cell *ij* represents the number of data/function/method references that exist from file *i* to file *j*. I refer to data references as data dependencies and function/method references as functional dependencies. A comparative analysis of the quarterly syntactic dependency information showed minimal variability (less than 0.5% across quarters) over time. Consequently, the information from the last quarter of each release covered by the data was used to compute all the syntactic dependency measures.

In the case of project A, a random sample of 100 files was selected to verify that the dependencies identified by the CREX tool were correct. The only problem encountered was missing dependencies in the cases of usage of function pointer, a traditional problem of most of the syntactic dependency identification tools (Murphy et al, 1998). Giving this particular problem, I searched for all the source code files of project A that used function pointers and a total of 279 files were identified. I manually updated the functional dependencies measures for that set of files. A similar analysis was performed on the dataset from project C.

I constructed an inflow, outflow and total count measure of both data and functional syntactic dependencies. Tables 9 and 10 report the pair-wise correlations of the various syntactic dependency measures with the other variables used in the statistical models. In order to select the appropriate set of syntactic dependency measures, two factors were evaluated: the predictive value of the measures and the pair-wise correlations to minimize collinearity problems. Based on those criteria, the inflow data dependency measure was selected.

Logical Dependencies: An alternative mechanism for identifying software dependencies considers the set of source code files that are modified together as part of a modification request as sharing technical dependencies. This approach was proposed by Gall and colleagues (1998) in the software evolution literature to identify logical dependencies between modules. One advantage of this approach is that it provides a better estimate for semantic dependencies relative to call graphs or data graphs because it does not rely on language constructs to establish the dependency relationship between source code files (Gall et al, 1998). The literature also refers to these software dependencies as evolutionary dependencies.

For both projects, the FCT method described in chapter 4 was used to construct the logical dependency matrix. In this study, the information from all the changes across all releases under consideration was accumulated in the logical dependency matrix. The data was accumulated because files that are changed together in a MR represent evidence of the existence of a logical dependency. The longer the period of time considered, the more changes take place, consequently, the accuracy of the identified logical dependencies increases.

Two file-level measures were extracted from the logical dependency matrix. First, the *Amount of Logical Dependencies* measure for file *i* was computed as the number of non-zero cells on column *i* of the matrix. The information in the diagonal of the matrix is ignored because it does not provide relational information. The diagonal just indicates the number of times a particular file was modified as part of modification requests. Since the matrix of logical dependencies is symmetric, this measure is equivalent to the degree of a node in undirected graphs terminology without considering self-loops.

A second measure, the *Clustering of Logical Dependencies* measure for file *i* was computed as the density of connections among the direct neighbors of file *i*. Unlike the amount of logical dependencies measure, this measure captures the nature of the interdependencies among the files that are interdependent with the focal file which is consistent with Herbsleb and colleagues' (2006) argument that the density of dependencies increases the likelihood of coordination breakdowns. This measure is equivalent to Watt's (1999) local clustering measure.

Work Dependencies: I constructed two different measures of work dependencies. A traditional measure, *Workflow Dependencies*, captures the temporal aspects of interdependencies in development tasks. For instance, sub-tasks of a particular development effort might need to be performed sequentially, therefore, imposing a temporal dependency, for instance, where sub-task A must be done before sub-task B can be started. In earlier chapters, I discussed the need for a richer measure of work dependencies that is able to capture the dynamism and complexity of software development work. In chapter 5, I showed that the coordination requirements measure has those attributes. Hence, it is also important to examine it impact on software quality.

Workflow Dependencies: Both projects used tools to track the progress of development tasks. The information stored in these tools provided the data necessary to construct the workflow followed by each modification request. A workflow is a traditional approach to identify work dependencies where two developers *i* and *j* are interdependent if the development task has to be transferred from developer *i* to developer *j* at some point in time. For instance, a modification request requires changes to two subsystems. Developer *i* completes the work on one of the subsystems and then he/she hands over the development task to developer *j* to finish the work on the second subsystem. Using the work dependency relationships between developers from all the modification requests considered in the study, I constructed a developer to developer matrix where a cell *ij* represents the number of work dependencies developer *i* has on developer *j*. In order to study the implications of the work dependencies on the development process, it is useful to think if those dependencies as a system of social relationships amongst developers. Then, social network analysis provides the methodological framework and the theoretical background to guide us in examining the implications of particular structures of those social relationships (Wasserman & Faust, 1993). One consistent result in the social network literature is that being centrally positioned in the system of social relationships has implications on the actor's ability to perform his/her tasks.

Although several measures have been developed to capture how central an individual is in a network, one measure has been widely explored: degree centrality (Freeman, 1979). Degree centrality attempts to identify central individuals based on the number of ties to other actors in the network. The idea is that more connections or ties benefits individuals because they provide numerous conduits to information and other resources. On the other hand, an increasing number of ties involves more effort dedicated to the communication. In the context of this study, increasing the number of ties involves augmenting the number of work dependencies associated with a developer. In turn, a developer requires increased effort and dedication to manage those relationships. Consequently, the quality of the produced software code could suffer. Mathematically, this idea of degree centrality is captured by the following formula:

$$DC(n_i) = \frac{d(n_i)}{n-1} \qquad \textbf{(Equation 5)}$$

Where *n* is the number of nodes in the network and $d(n_i)$ is the number of connections of node n_i, The values of this measure range from 0, which indicates the node is an isolate or it is not connected to any other node, to 1 which indicates that a particular node *i* has a ties to all other *n - 1* nodes. The degree centrality measure described above was used to relate the impact of work dependencies on the failure proneness of a source code file.

The *Workflow Dependencies* measure was constructed in the following way. For each file, I identified the developer that worked on the file that had the highest value of degree centrality in the workflow network. Using the maximum value of degree centrality is based on the idea that a single highly constrained developer that modified a particular file could be sufficient to introduce a defect into the system. An alternative approach would be to compute this measure as an average of the degree centrality of the set of developers that worked on each particular file. However, the number of developers that modified a file was highly correlated with other measures. Hence, it was more appropriate to use the measure based on the maximum value of degree centrality and network constraint measures.

Coordination Requirements: A developer by developer matrix was constructed using equation 1 as it was described in chapter 2. The T_A matrix was built using the information from the MR reports. The logical dependency matrix described in previous paragraphs constituted the T_D matrix. Then, the developer to developer linkage information was related to the files using the same procedure used to compute the workflow dependencies measure. In other words, for each file, I identified the developer that worked on the file that had the highest value of degree centrality in the coordination requirements matrix.

Congruence measures: In order to construct these measures, I used the same procedure described in chapter 5. Since coordination data over IRC was only available for a subset of the modification requests considered in this study, I computed only structural, geographical and MR congruence measures for each modification request.

The congruence measures at the file level were constructed in the following way. For each file, I identified all the modification requests that touched the file. Then, the median value of each congruence measure was associated with that particular file. An alternative approach would be to compute the measure as an average of level of congruence across all modification requests that affected each particular file. However, the number of MRs that affected a file was highly correlated with other measures and it was also a significant predictor of failures. Hence, it was more appropriate to use the measure based on the median value of congruence.

Additional Predictors: The objective of the study is to examine the relative impact that important conceptual factors such as technical and work dependencies have on failure rather than improving existing predictive model of failures. However, the analysis cannot neglect those factors that past research have found associated to failures. Over the past decades, numerous measures have been used to predict failures (see for instance Graves et al, 2000; Fenton & Neil, 1999; Nagappan & Ball, 2007; Ostrand et al, 2005). As suggested by Graves and colleagues (2000), those measures could be grouped in two sets: process measures and product measures. Process measures such as number of previous faults, number of deltas, age of the code and the number of developers that modified the files have been shown to be very good predictors of failures (Graves et al, 2000; Nagappan & Ball, 2007). These measures are also referred to in the literature as churn metrics. In this study, I measured the number of MRs, which is similar to the number of past failures a file had, as well as the average number of lines changed in a particular file as part of modification requests. One problem with the process measures is they tend to be highly correlated with other measures (Nagappan & Ball, 2007), and this study was not an exception. Tables 11 and 12 show that the *number of MRs* measure is highly correlated with *LOC*, *Average Lines Changed* and with the measures of logical dependencies, particularly in project C. In order to minimize collinearity problems, only *LOC* and *Average Lines Changed* were used in the models.

On the other hand, the results for product measures such as size of the code and complexity measures are mixed. Researchers have found a positive relationship between lines of code and failures (Briand et al, 2000; Graves et al, 2000). However, other work has found a negative relationship between lines of code and failures, that is, a larger number of LOC decreases the likelihood of failures (Basili & Perricone, 1984). In this study, the size of the module was measured as the number of non-blank non-comment lines of code.

Table 9: Descriptive Statistics for Last Release of Project A

	Mean	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
FileBuggyness	0.458	0.498	0	1	0.167	1.028
LOC	496.8	919.1	23	17853	6.090	71.51
Avg. Lines Changed	12.19	39.35	0	671	8.171	92.35
Syntactic Dep.	5.158	66.56	0	1741	21.82	509.8
Num. Logical Dep.	102.6	114.3	0	836	1.883	7.525
Clustering Logical Dep.	0.751	0.284	0	1	-0.996	3.155
Workflow Dep.	0.227	0.115	0	0.386	-0.174	1.728
Coordination Req.	0.137	0.121	0	0.623	2.655	11.91
Structural Congruence	0.151	0.271	0	0.376	1.012	5.939
Geo. Congruence	0.231	0.304	0	0.476	-3.362	6.943
MR Congruence	0.152	0.205	0	0.584	2.048	2.933

Table 10: Descriptive Statistics for Last Release of Project C

	Mean	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
FileBuggyness	0.103	0.305	0	1	2.598	7.753
LOC	838.2	3515.1	16	65542	16.09	288.1
Avg. Lines Changed	18.14	55.99	0	949	10.79	154.5
Syntactic Dep.	42.69	170.2	0	1979	8.177	83.16
Num. Logical Dep.	23.95	26.09	0	233	2.916	16.61
Clustering Logical Dep.	0.220	0.239	0	1	2.013	6.618
Workflow Dep.	0.347	0.142	0.010	0.704	-0.133	2.725
Coordination Req.	0.814	0.188	0	0.976	-2.067	7.677

Table 11: Pair-wise Correlations for Last Release of Project A (* p < 0.01)

	1	**2**	**3**	**4**	**5**	**6**
1.FileBugyness	-					
2.LOC (log)	0.282*	-				
3.Number MRs (log)	0.371*	0.241*	-			
4.Avg. Lines Chg. (log)	0.185*	0.274*	0.303*	-		
5.In-Data Dep. (log)	0.065*	0.001	0.079*	0.033	-	
6.Out-Data Dep. (log)	0.183*	0.479*	0.190*	0.191*	-0.263*	-
7.Total-Data Dep. (log)	0.222*	0.448*	0.235*	0.199*	0.344*	0.796*
8.In-Funct. Dep. (log)	0.041*	0.271*	0.090*	0.091*	-0.099*	0.370*
9.Out-Funct. Dep. (log)	0.118*	0.433*	0.149*	0.159*	-0.240*	0.778*
10.Total-Funct. Dep. (log)	0.086*	0.416*	0.135*	0.142*	-0.218*	0.702*
11.Num. Logic. Dep. (log)	0.491*	0.335*	0.454*	0.161*	0.043*	0.233*
12.Cluster Logic. Dep. (log)	-0.322*	-0.217*	-0.293*	-0.132*	-0.056*	-0.167*
13.Workflow Dep. (log)	0.336*	0.078*	0.378*	0.122*	0.019	0.073*
14.Coordination Req. (log)	0.244*	0.095*	0.408*	0.148*	0.025	0.070*
15.Structural Cong (log)	0.161*	0.180*	0.392*	0.141*	0.037	0.153*
16.Geo. Cong. (log)	0.111*	0.182*	0.314*	0.124*	0.042	0.119*
17.MR Cong. (log)	0.177*	0.187*	0.418*	0.104*	0.012	0.095*

	7	8	9	10	11	12
7.Total-Data Dep. (log)	-					
8.In-Funct. Dep. (log)	0.292*	-				
9.Out-Func. Dep. (log)	0.596*	0.436*	-			
10.Total-Funct. Dep. (log)	0.538*	0.736*	0.892*	-		
11.Num. Logic. Dep. (log)	0.258*	0.061*	0.188*	0.163*	-	
12.Cluster Logic. Dep. (log)	-0.202*	-0.099*	-0.138*	-0.126*	-0.052*	-
13.Workflow Dep. (log)	0.103*	-0.063*	-0.018	-0.041*	0.348*	-0.135*
14.Coordination Req. (log)	0.092*	-0.013	0.007	-0.004	0.266*	-0.164*
15.Structural Cong (log)	0.173*	0.148*	0.154*	0.161*	0.223*	-0.242*
16.Geo. Cong. (log)	0.115*	0.113*	0.097*	0.101*	0.322*	-0.197*
17.MR Cong. (log)	0.108*	0.163*	0.076*	0.150*	0.305*	-0.151*

	13	14	15	16
13.Workflow Dep. (log)	-			
14.Coordination Req. (log)	0.257*	-		
15.Structural Cong (log)	0.157*	0.212*	-	
16.Geographical Cong. (log)	0.219*	0.187*	0.313*	-
17.MR Cong. (log)	0.301*	0.183*	0.249*	0.282*

Table 12: Pair-wise Correlations for Last Release of Project C (* p < 0.01)

	1	**2**	**3**	**4**	**5**	**6**
1.FileBugyness	-					
2.LOC (log)	0.269*	-				
3.Number MRs (log)	0.502*	0.413*	-			
4.Avg. Lines Chg. (log)	0.168*	0.422*	0.346*	-		
5.In-Data Dep. (log)	0.129*	0.269*	0.155*	0.136*	-	
6.Out-Data Dep. (log)	0.211*	0.599*	0.295*	0.376*	0.081*	-
7.Total-Data Dep. (log)	0.195*	0.583*	0.293*	0.346*	0.599*	0.759*
8.In-Funct. Dep. (log)	0.171*	0.370*	0.233*	0.199*	0.225*	0.528*
9.Out-Funct. Dep. (log)	0.239*	0.600*	0.374*	0.389*	0.119*	0.890*
10.Total-Funct. Dep. (log)	0.218*	0.542*	0.362*	0.358*	0.107*	0.848*
11.Num. Logic. Dep. (log)	0.272*	0.245*	0.631*	0.251*	0.109*	0.192*
12.Cluster Logic. Dep. (log)	-0.218*	-0.143*	-0.322*	-0.089*	-0.094*	-0.153*
13.Workflow Dep. (log)	0.285*	0.149*	0.400*	0.049	0.169*	0.122*
14.Coordination Req. (log)	0.174*	0.023	0.278*	-0.060	0.159*	0.032

	7	8	9	10	11	12
7.Total-Data Dep. (log)	-					
8.In-Funct. Dep. (log)	0.379*	-				
9.Out-Funct. Dep. (log)	0.669*	0.557*	-			
10.Total-Funct. Dep. (log)	0.603*	0.785*	0.916*	-		
11.Num. Logic. Dep. (log)	0.238*	0.109*	0.201*	0.203*	-	
12.Cluster Logic. Dep. (log)	-0.132*	-0.184*	-0.187*	-0.190*	0.161*	-
13.Workflow Dep. (log)	0.146*	0.140*	0.164*	0.143*	0.287*	-0.186*
14.Coordination Req. (log)	0.092*	0.124*	0.087*	0.090*	0.207*	-0.144*

	13
13.Workflow Dep. (log)	-
14.Coordination Req. (log)	0.611*

Results

The analysis was organized in three stages. First, I focused on examining the relative impact of each of the types of dependencies on failure proneness of source code files. The data corresponding to the last release from each project was used in this analysis. Secondly, a stability analysis across all the releases of both projects was performed to verify the consistency of the results found in the first step. Finally, additional confirmatory analysis was done considering the data from all the releases of each project into a single longitudinal model.

The Impact of Dependencies

Several logistic regression models were constructed to examine the impact of each class of independent variables on failure proneness of a software system using the data from the last release of each project. I started the analysis with a baseline model that contains only traditional predictors. In subsequent models, I added the measures for syntactic, logical and work dependencies as well as congruence described in the previous section. It is important to assess the fitness of the model to evaluate whether each measure in fact has a tangible impact on failure. Therefore, for each statistical model I report the Akaike's Information Criteria (AIC) and the percentage of deviance explained by the model. The AIC is an indicator of the fit and the explanatory power of the model. Lower values of the AIC indicate better fit of the model to the data. The AIC also adjust for the number of variables in the model. Hence, as more independent variables are added to the model, the AIC would decrease if those additional variables do not have a tangible contribution to the explanatory power of the model. Logistic regressions were estimated with a maximum-likelihood method and the deviance is defined as -2 times the log-likelihood of the model. The percentage of the deviance explained compares the log-likelihood of the null model with the log-likelihood of the full model. Using the AIC and the percentage of deviance explained, I can compare the contribution that each independent variable has on explaining the variance exhibited by our dependent variable *FileBuggyness*.

The first model is a baseline model which includes the size of the module in lines of code (LOC) and the average number of lines changed as predictors. Table 13 shows the estimated coefficients of the logistic regression. In both projects, *LOC* is positively associated with failure proneness. These results agree with those found by Briand and collegues (2000). Past research has found conflicting results relating LOC to failures. Two studies (Basili & Perricone, 1984; Moeller & Paulish, 1993) found evidence suggesting that a larger file tends to have lower defect densities possibly because larger modules tend to be developed more carefully (Moeller & Paulish, 1993). The measure of *Average Lines Changed* is positively related to failure proneness in both projects suggesting the higher the amount of modification that occurs in a source code file, the higher the likelihood of encountering field defects associated with that file. Finally, based on the values of the AIC, the baseline model has a much better fit in the case of project C (lower AIC) relative to project A. Also, the two traditional predictors explain almost 50% more deviance in project C relatively to the model for project A.

Table 13: Baseline Model for Failure Proneness

	Project A	**Project C**
Intercept	-2.158**	-6.471**
LOC (log)	0.333**	0.641**
Avg. Lines Changed (log)	0.189**	0.171*
AIC	5100	725
Deviance Explained	7.2%	11.8%
Nagelkerke R^2	0.126	0.156

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

The second model includes the syntactic dependency measure. Table 14 shows the results of the logistic regression. Consistent with previous research, syntactic dependencies increase the likelihood of failure on the source code. It is important to highlight that syntactic dependencies provide a small contribution to the explanatory power of the model, particularly in project A. However, its impact is still statistically significant. As described earlier in the chapter, several measures of syntactic dependencies were evaluated (inflow/outflow data and functional syntactic dependencies). The ones used in this regression model were inflow data dependencies for both projects. That choice was based on the contributions of those particular measures to the explanatory power of the model as well as minimizing collinearity problems.

Table 14: Impact of Syntactic Dependencies on Failure Proneness

	Project A	**Project C**
Intercept	-2.213**	-6.499**
LOC (log)	0.336**	0.611**
Avg. Lines Changed (log)	0.185**	0.167*
Syntactic Dep. (log)	0.162**	0.112*
AIC	5084	722
Deviance Explained	7.5%	12.4%
Nagelkerke R^2	0.132	0.164

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

In the next step, I included the two measures of logical dependencies in the model. Table 15 shows the results. Higher number of logical dependencies increases the likelihood of failure as expected. The pair-wise correlations reported in tables 11 and 12 showed relatively low levels of correlation between syntactic and logical dependency measures. Those results combined with the ones reported on table 15, suggests the effect of logical dependencies on failure proneness is complementary to the impact of syntactic dependencies. The impact of clustering of logical dependencies reduces the likelihood of failures suggesting that as clusters of interrelated files emerge, developers might become more cognizant of such relationships and, consequently, they increase their effort making sure that changes to the system do not introduce additional problems. The contribution of the logical dependencies measures to the explanatory power of the model is quite important in both projects, particularly, in relation to the contribution of the traditional syntactic dependency measures.

Table 15: Impact of Logical Dependencies on Failure Proneness

	Project A	Project C
Intercept	-1.569**	-5.906**
LOC (log)	0.102**	0.498**
Avg. Lines Changed (log)	0.133**	0.044
Syntactic Dep. (log)	0.104*	0.029
Number Logical Dep. (log)	0.831**	0.780**
Clustering Logical Dep. (log)	-4.700**	-3.902**
AIC	3868	621
Deviance Explained	29.7%	25.2%
Nagelkerke R^2	0.450	0.319

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

Interestingly, in project C the measure of average lines changed loses statistical significance once syntactic dependencies enters the model. One possible explanation for this result could be related to the level of correlation between *Average Lines Changed* and the variables *LOC* (0.4223) and *Syntactic Dependencies* (0.3364). Unfortunately other churn metrics such as average lines of code added or number of previous faults were even more correlated with those variables, hence, the selection of *Average Lines Changed*.

Next, the measures of work dependencies are introduced into the model. First, I examined the impact of workflow dependencies. Table 16 shows consistent results across projects. The higher the amount of workflow dependencies developers that modified a file has, the higher the likelihood of source code files to be associated with field defects. As expected, these results suggest when developers become more constrained by their work dependencies, they become more prone to make mistakes and, consequently, introduce defects into the software system. Table 17 shows the impact of the second work dependency measure: coordination requirements. This work dependency measure combines the logical dependency information and data regarding which developers participated in the effort associated with each modification request in order to determine which developers should coordinate with. In project A, the coordination requirement measure also increases the likelihood of failures. However, in project C, its effect is not statistically significant. This is not surprising given the high correlation between the two work dependency measures (0.661) found in project C (see table 12).

Table 16: Impact of Workflow Dependencies on Failure Proneness

	Project A	**Project C**
Intercept	-2.259**	-8.664**
LOC (log)	0.127**	0.508**
Avg. Lines Changed (log)	0.117**	0.071
Syntactic Dep. (log)	0.102*	0.001
Number Logical Dep. (log)	0.742**	0.555**
Clustering Logical Dep. (log)	-4.464**	-2.695**
Workflow Dep. (log)	4.099**	8.297**
AIC	3783	588
Deviance Explained	31.3%	29.6%
Nagelkerke R^2	0.469	0.368

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

In sum, the previous analysis showed that syntactic, logical and work dependencies impact failures in a software system. More importantly, their role is complementary suggesting the various types of dependencies capture different relevant aspects of the technical properties of a software system as well as elements of the software development process.

Table 17: Impact of Coordination Requirements on Failure Proneness

	Project A	**Project C**
Intercept	-2.367**	-10.03**
LOC (log)	0.134**	0.511**
Avg. Lines Changed (log)	0.107**	0.075
Syntactic Dep. (log)	0.101*	-0.008
Number Logical Dep. (log)	0.724**	0.545**
Clustering Logical Dep. (log)	-4.369**	-2.560**
Workflow Dep. (log)	3.867**	7.370**
Coordination Requirements (log)	2.156**	2.609
AIC	3771	588
Deviance Explained	31.5%	29.8%
Nagelkerke R^2	0.472	0.370

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

Finally, I examined the impact of congruence measures on failure. Coordination activity data was not available for project C, so Table 18 only reports results associated with project A. As expected, when developers coordinate their work appropriately, the likelihood of failures is reduced. However, unlike the productivity study (chapter 5), the results showed that only MR congruence is relevant. Structural and geographical congruence did not reach statistical significance. These results suggest that in the case of project A, the coordination activities over the MR tracking system were critical in terms of quality relative to other means of communication and coordination.

Table 18: Impact of Congruence on Failure Proneness

	Project A
Intercept	-2.136**
LOC (log)	0.141**
Avg. Lines Changed (log)	0.112**
Syntactic Dep. (log)	0.109*
Number Logical Dep. (log)	0.758**
Clustering Logical Dep. (log)	-4.522**
Workflow Dep. (log)	4.237**
Coordination Requirements (log)	2.191**
Structural Congruence (log)	-14.41+
Geographical Congruence (log)	-0.947
MR Congruence (log)	-3.888**
AIC	3759
Deviance Explained	31.9%
Nagelkerke R^2	0.474

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

Stability Analysis

The previous section showed that the different types of dependencies as well as congruence impacted failure proneness in the last release covered by the data in both projects. Now, I turn the attention to examining the robustness of those results across all the releases in each project. In the case of project A, the measures of congruence are also included in the models. In the case of project C, given the high correlation between workflow dependencies and coordination requirements (avg=0.6032, min=0.3736, max=0.6869), I opted for including only workflow dependencies in the models. Table 19 reports the coefficients for all the measures from the logistic regression using the data from the three remaining releases of project A. Overall, the results are consistent with those reported in table 16. There are a few exceptions. First, the traditional predictors, *LOC* and *Average Lines Changed*, do not show consistent results across releases. This is not particularly surprising giving the inconsistency of results reported in past research, particularly in relation to the impact of LOC (Fenton & Neil, 1999). A second exception is the first release, where workflow dependencies and congruence also lack statistically significant effects. A possible explanation for these results is the difference in the nature of the development work across releases. In the first release of project A, most of the development work is dominated by feature development. Then, the amount of workflow dependencies is lower because each modification request represents larger development tasks. As the product matures, the development work involved more defect resolution and less feature development. This situation would increase the workflow dependencies and the importance of using coordination tools such as the MR tracking system.

Table 19: Impact of Technical Dependencies, Work Dependencies and Congruence across Releases in Project A

	Release 1	Release 2	Release 3
Intercept	2.987	2.809	-2.216**
LOC	0.015	0.022	0.114**
Avg. Lines Changed	0.064	-0.029	-0.021
Syntactic Dep.	0.176*	0.171**	0.102+
Number Logical Dep.	0.977**	0.876**	0.834**
Clustering Logical Dep.	-5.124**	-4.437**	-4.017**
Workflow Dep.	0.648	1.234*	5.497**
Coordination Requirements	7.176*	2.892**	4.670+
Structural Congruence	-11.36	-13.71	-11.08
Geographical Congruence	-1.480	-1.249	-0.822
MR Congruence	-0.477	-1.377**	-7.920**
AIC	2462	3020	3440
Deviance Explained	34.6%	30.1%	32.2%
Nagelkerke R^2	0.516	0.489	0.495

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

Table 20 shows the results for the remaining five releases of project C. In this case, the results are very consistent across all releases confirming the original findings reported in table 16. Interestingly, the explanatory power of the model is about 50% higher for the first two releases of project C. It is possible that over time, the system structure evolves increasing the importance of other factors such as the ability of developers to coordinate their work in a congruent fashion. Unfortunately, the lack of coordination activity data from project C does not allow me to evaluate such a claim.

Table 20: Impact of Technical Dependencies, Work Dependencies and Congruence across Releases in Project C

	Release 1	**Release 2**	**Release 3**
Intercept	-7.258**	-9.615**	-7.298**
LOC	0.673**	0.681**	0.626**
Avg. Lines Changed	-0.214	-0.465	-0.291*
Syntactic Dep.	-0.003	0.058	-0.021
Number Logical Dep.	1.124*	1.286*	0.795**
Clustering Logical Dep.	-6.284*	-5.012**	-3.241**
Workflow Dep.	2.563*	5.695**	4.195**
AIC	118	145	437
Deviance Explained	42.4%	47.3%	28.8%
Nagelkerke R^2	0.523	0.543	0.367
	Release 4	**Release 5**	
Intercept	-7.111**	-8.060**	
LOC	0.480**	0.512**	
Avg. Lines Changed	-0.015	0.036	
Syntactic Dep.	-0.016	-0.007	
Number Logical Dep.	0.606**	0.625**	
Clustering Logical Dep.	-3.123**	-3.028**	
Workflow Dep.	6.310**	7.003**	
AIC	545	566	
Deviance Explained	27.8%	28.9%	
Nagelkerke R^2	0.355	0.365	

(+ $p < 0.10$; * $p < 0.05$; ** $p < 0.01$)

Checks for Random Temporal Effects

The final step of the analysis consisted of considering all releases into a single model. This approach allows for controlling any random effects associated with the passage of time or effects that are specific to each release, providing an additional confirmatory test of the results reported in tables 16 and 17. The longitudinal nature of this analysis breaks the assumptions of a traditional linear or logistic regression statistical model. The dataset contains multiple observations of the same source code file which results in lack of independence of the observations. In order to correctly deal with that lack of independence, I used a random effects logistic model to examine the effect of the various class of dependencies on failure proneness. Table 21 reports the results. Overall, the results are consistent with those reported in table 17, confirming the reliability of the effects and the findings reported earlier.

Table 21: Random-effects Model of Failure Proneness

	Project A	**Project C**
Intercept	-2.903**	-13.32**
LOC	0.131**	0.933**
Avg. Lines Changed	0.057	-0.121
Syntactic Dep.	0.259**	0.014
Number Logical Dep.	1.939**	1.409**
Clustering Logical Dep.	-10.94**	-7.221**
Workflow Dep.	2.083**	7.586**
Coordination Requirements	0.555*	-
AIC	6277	1063
Deviance Explained	27.7%	31.9%

(+ p < 0.10; * p < 0.05; ** p < 0.01)

Discussion

The study reported in this chapter has several important contributions to the software engineering literature. First, the study examined the impact that syntactic, logical and work dependencies have, simultaneously, on the failure proneness of a software system. All three types of dependencies are relevant and their effect is complementary suggesting their independent and important role in the development process. Consistent with past results, the analysis showed that source code files with higher number of syntactic dependencies were more prone to failure. More importantly, the results also showed that source code files with higher number of logical dependencies are more likely to exhibit field defects. In addition, this study is the first analysis that highlights the importance of the structure of the logical relationships. The results showed that software modules with logical dependencies to other highly interconnected files were less likely to exhibit customer-reported defects. Then, this finding suggests a new view of product dependencies with significant implications regarding how we think about modularizing the system and how development work is organized. The effect of the structure of the network of product dependencies elevates the idea of modularity in a system to the level of "clusters" of source code files. Then, those highly inter-related sets of files become the relevant unit to consider when development tasks and responsibilities are assigned to organizational groups.

A second significant contribution of the study reported in this chapter is the recognition and the assessment of the impact the engineers' social network has on the software development process. The results showed that individuals that exhibited a higher number of workflow dependencies and coordination requirements were more likely to introduce defects in the files they worked on. These findings suggest the potentially detrimental effect of the additional effort on the part of a developer that needs to receive work from or coordinate with multiple people and manage those relationships accordingly in order to perform the tasks.

Finally, the study has two additional important contributions. The empirical analysis was replicated across two distinct projects from two unrelated companies obtaining consistent results. Then, this study exhibits strong external validity, a factor typically neglected in the software engineering literature. In addition, the statistical models proposed in this chapter showed significantly higher level of predictive power than recent proposed models of failure proneness (Nagappan & Ball, 2007) that focused on the role of traditional factors such as syntactic dependencies and churn metrics.

CHAPTER 7: THE EVOLUTION OF COORDINATION BEHAVIOR

Over the past couple of decades, geographically distributed work has become pervasive and product development organizations are no exception. Unfortunately, this new trend has its costs. It is well established that physical proximity facilitates interactions among individuals working in R&D organizations (e.g. Allen, 1977; Herbsleb & Mockus, 2003). Distance leads to numerous problems in communication and coordination, and ultimately, impacts the performance of product development teams (Brown & Eisenhardt, 1995; Herbsleb & Mockus, 2003; McDonough et al, 2001). A reduction in communication has been linked to failure to identify dependencies among work teams resulting in coordination problems (de Souza et al, 2004; Grinter et al, 1999; Herbsleb & Mockus, 2003; Yassime et al, 2003). The results reported in previous chapters showed that neglecting coordination needs have detrimental effect on software development and quality of the software system produced. In chapter 5, I showed that more productive developers tend to coordinate their work more effectively highlighting the importance of social factors in software development. In order to support distributed teams appropriately, it is important to understand how information flows among teams and across sites, and the characteristics of the individuals that occupy key roles in the communication network. In this chapter, I present a longitudinal examination of coordination patterns among developers using data collected from project A. In addition, data from project D was used to replicate a portion of the statistical analysis that explored the relationship between coordination patterns and individual-level productivity.

Study III: The Evolution of Coordination Behavior

March and Simon (1958) argued that tasks should be divided into nearly independent parts and when interdependence is unavoidable, appropriate coordination mechanisms should be put in place. In the context of product development organizations, there is a close relationship between dividing development tasks into nearly independent parts and partitioning the system to be developed into nearly independent parts. Modularization is the approach typically used to minimize technical dependencies among the parts of a system (Conway, 1968; Eppinger et al, 1994; Sullivan et al, 2001). As I argued earlier in this dissertation, the modular design approach has important limitation when applied to the context of geographically distributed software development. Those limitations suggest that communication among teams will be essential in order to coordinate project work. Organizational and geographic barriers to communication can be overcome by individuals in key roles who facilitate and promote the interaction between teams (Allen, 1977; Ancona & Caldwell, 1992; Hauschildt & Schewe, 2000). Several definitions of those key positions have been proposed in the product development literature (Hauschildt & Schewe, 2000). Examples are “alliance champion”, “external liaison”, “gatekeeper”, and “process promoter”. Although those definitions differ slightly from each other in their theoretical underpinnings, the overarching theme is that those individuals perform a different type of activity than the rest of the members of a R&D group and their task is critical for the success of a project. Those key people have access to different sources of information and they are capable of synthesizing the information in a way useful for the various groups so they can to better perform their development activities (Hauschildt & Schewe, 2000).

The use of “liaison” or “gatekeepers” to manage the dependencies between teams has also been proposed as a mechanism for facilitating coordination in geographically distributed software development (Sangwan et al, 2006). As engineers perform their development tasks, critical information and knowledge about the parts of the system involved in the tasks at hand is exchanged. As software development tasks change over time, developers get the opportunity to gain access to new information and knowledge about the technical properties of different parts of the system. This system of social relationships, which I will refer to as a *Coordination Network*, is an evolving entity. If gatekeepers are strategically embedded in the coordination networks, they can acquire the necessary knowledge to discover the relevant technical and task dependencies.
This study attempts to shed light on how coordination networks evolve in a geographically distributed software development projects in order to address the limits of the design modularity strategy.

Research Questions

Past research on communication patterns in R&D organizations (Allen, 1977; Hauschildt & Schewe, 2000) suggests that a “gatekeeper” type of communication network will emerge:

RQ1: Does a relatively small group of people take on a disproportionate share of overall communication?
RQ2: Does a relatively small group of people take on a disproportionate share of cross-team, cross-site communication?

Gatekeepers in R&D organizations are also perceived as very technically competent individuals who are able to interpret several sources of information, translate them and synthesize them to be consumed by development teams (Allen, 1977; Hauschildt & Schewe, 2000). Work in social networks argues that maintaining connections involves important amounts of energy (e.g. Burt, 1992). Then, individual-level contributions to the project, in terms of direct labor, might be detrimentally affected. It is also important to understand the characteristics of the people who assume a gatekeeper role in software development organizations.

RQ3: Are the most productive technical people part of the core of the coordination networks?

RQ4: What other characteristics can lead to a technical person becoming part of core the coordination networks?

As tasks are performed in organizations, communication channels emerge. Over time, organizations also develop filters that identify the most relevant information pertinent to the task at hand (Daft & Weick, 1984). In other words, organizations develop stable communication and coordination patterns. If the task dependencies of the product development effort change, those established information flows and filters might become inadequate and, consequently, disrupt the organization's ability to coordinate effectively. For example, Henderson & Clark (1990) found that minor changes in product architecture can generate substantial changes in task dependencies, and drastically affect the organizations' ability to coordinate work.

RQ5: How stable are the communication roles and positions in the coordination networks over time?

On a related issue, the studies that examined drivers of communication in product development organizations (e.g. Morelli et al, 1995; Sosa et al, 2004) argue that the technical dependencies between parts of system developed by different organizational units tend to be a main driver of interactions between those organizational entities. Since identifying dependencies in software development is more challenging than in many other types of product development efforts, I seek to understand if the findings from the product development literature will hold in the context of software development organizations. Therefore, the evolution of patterns in the coordination networks is also a concern, in particular:

RQ6: Do coordination requirements drive communication patterns?

Method

The data associated with 2375 multi-group modification requests from project A was used to examine the research questions presented in the previous section. The data covered the development effort of the first four releases of the product.

Description of the Data

Software developers communicated and coordinated their development tasks using various means of communication. Opportunities for interaction exist when individuals work in the same formal team or in the same location. For instance, all the development teams had periodic meeting as frequent as once or more times a week. Developers also used a range of communication tools to interact and coordinate their work such as email, IRC, video conference, and the MR tracking system. I met with several developers, who identified IRC as the primary communication means for development and debugging work. The second most commonly used tool was the MR tracking system. Developers also used email and video-conferences primarily for design and architectural definition type of activities. Given those patterns of communication means usage, communication and coordination information was collected from IRC logs and the MR tracking system.

On a daily basis, developers interacted with other engineers in the same or other laboratories using IRC. The company established several channels based on formal teams as well as special projects. For example, team name "A" is responsible for components 1 and 2, then there is a channel name "A" in IRC. Any engineer that requires information about components 1 and 2 would typically communicate with other engineers in channel "A". In order to preserve the valuable technical information discussed on IRC, the company logged the channels associated with formal teams and special projects. This repository provided historical data that allowed me to reconstruct the patterns of interaction and coordination amongst the developers. The set of MRs guided the identification of the relevant interactions. Three raters, blind to the research questions, examined the IRC logs corresponding to all the recorded channels. Since the work on a MR could extend over days, weeks or even months, the raters were instructed to examine IRC logs through out the entire period of time associated with each MR. When interacting, developers could refer to the MR id number (e.g. "<developer01> developer02: have you looked at bug 12345") or to the problem the MR represents without any explicit reference to the MR (e.g. "<developer01> does anyone know why would RPC call 123 returns the error code 12345?"). The raters were given a description of the problem associated with each modification request in other to be able to identify the latter type of interactions. I assessed the reliability of the raters' work by having them code 10% of the MRs by all three raters. Comparisons of the obtained networks showed that 98.2% of the networks had the same set of nodes and edges. Based on that data, I constructed the coordination networks on a monthly basis. The networks contain all one hundred and fourteen developers. If any of the developers did not participate in any discussion on the IRC logs for a particular month, he or she would be represented in the network as a node without connections, in other words, an isolate.

The company also used a MR tracking system to monitor the progress of development tasks and to facilitate the exchange of information and discussion about the development tasks. For example, as defects are debugged, developers post information regarding their findings and might request information from other developers that would provide useful feedback. I defined an interaction between developers *i* and *j* only when both *i* and *j* explicitly commented in the MR report. The focus was on the developers that explicitly commented on the MR report because the MR tracking system sent email to all the addresses in a CC list every time an MR is updated. Therefore the recipients of updates could be significantly larger than the set of people actually providing information to the MR. Comments automatically generated by the workflow tool were also ignored (e.g. changes to the status of the task). Then, I used the exchanges of information to construct coordination networks amongst developers. In this case, the data collection process was automated by using a script that interacted with the modification request tracking system and constructed the monthly social networks.

Description of Measures

I computed several individual level measures such as individual-level performance, traditional factors that have been found to predict development performance (e.g. programming and domain experience) and network measures that capture different structural properties of the individuals' position in the coordination networks.

Individual-level Performance: Measuring individual-level performance in software development is a challenging task. The concept of performance could be interpreted across different dimensions such as the amount of code produced, the quality of that produced code in terms of lack of defects, efficiency and maintainability as well as the adherence of the system's functionality to the requirements. Previous research had taken different approaches and each one has its benefits and drawbacks. The pioneering work on programmers' productivity (Curtis, 1981) focused on the amount of code produced and its relationship with the cognitive ability and programming experience of developers. The "amount of code" measures such as SLOC are programming language dependent so comparisons across projects are not feasible. If a project involves significant portions to be developed in significantly different programming languages, a SLOC-type of measure is also problematic. On the other hand, the information systems literature tends to focus on measures of performance collected through self-assessment questionnaires or from managerial ratings records (Rasch & Tosi, 1992). In both cases, the measures are subjective, however, they could be used to draw comparisons across projects.

The project under analysis in the study was mostly developed in the same programming language (C language). There was a small portion of the system developed in C++ language, however, it was a module of the system that ran in the kernel so the coding style was very similar to a program written in the C language. Given these characteristics of the project, I used two sources of performance data. First, a measure of contribution to the development effort is defined in terms of amount of code produced, *NumChanges*. A measure based on number of changes, instead of a more traditional lines-of-code measure, allows us to control for variability in developers' coding style (e.g. developers who might have a more verbose versus a more compact coding style). Moreover, the development organization studied encouraged developers to submit changes to the version control system that constituted logical pieces of work as a single commit. Hence, the measure *NumChanges* represents an appropriate measure of task performance. A second measure of performance is represented by the number of modification requests resolved by each developer, *NumMRs*. In the dataset, both performance measures were highly correlated because all changes to the source code were represented by a modification request.

Network Measures: Over the years, numerous measures have been proposed to capture different aspects of the structure of the social networks and the individuals' position within those networks. In this analysis, I selected network measures that have been empirically examined and previous research has found to have a direct relationship with individual-level performance. The differences among these measures are subtle but important. They suggest different strategies an individual might take toward constructing an effective communication network, and they also differ in the extent to which they view this process as cooperative or competitive. The ORA program (Carley & De Reno, 2006) was used to compute the network measures described below out of the monthly coordination networks.

Degree centrality (Freeman, 1979): The simplest definition of actor centrality is that central individuals must be the most active in the sense that they have the most ties to other actors in the network. The idea is that more connections or ties benefits individuals because they provide numerous conduits to information and other resources. Mathematically, this idea is captured by the formula described in equation 5 (Chapter 6).

Eigenvector centrality (Bonacich, 1987): Bonacich proposed a measure to assess the degree to which an individual's status is a function of the status of those to whom he or she is connected. This measure builds on the idea of degree centrality, but seeks to identify not only the individuals that have numerous ties but also those individuals that have numerous ties to other individuals who are also highly connected. Then, it could be argued that the benefits stemming from access to information (and other resources) are augmented because the ties to already resourceful individuals. The formulation presented by Bonacich (1987) for computing the eigenvector centrality of node *i* is the following:

$$\lambda e_i = \sum_j R_{ij} e_j \quad or \quad \lambda e = Re$$ **(Equation 6)**

Where, R is the matrix of relationships, *e* is eigenvector of R and λ is a constant for the equations to have a non-zero solution, in other words, the eigenvalue associated with *e*. The values of this measure range from 0 to 1.

<u>*Betweenness centrality*</u> (Freeman, 1977): This view of centrality diverges from degree and eigenvector centralities because it focuses on control of the flow of information rather than just access to information. More specifically, this measure examines the role of a node in the network by considering the probability that a communication from actor j to actor k takes a particular route. The betweenness measure proposed by Freeman (1977) assumes that the lines have equal weight and that communications will travel along the shortest route, hence the geodesics are considered in the following formula:

$$BC(n_i) = \frac{\sum_{j<k} \frac{g_{jk}(n_i)}{g_{jk}}}{[n-1][n-2]/2}$$ **(Equation 7)**

Where *n* is the number of nodes in the network, g_{jk} is the number of geodesics or shortest path between nodes *j* and *k*, and $g_{jk}(n_i)$ refer to the number of shortest paths between *j* and *k* that include node *i*. Then, $g_{jk}(n_i)/ g_{jk}$ is the probability of node *i* being in "between" in the communication between *j* and *k*. The denominator is the sum of probabilities over all pairs and it ranges from 0 when node *i* is not part of any geodesic to a maximum of (n-1)(n-2)/2 which accounts for all pairs of nodes not including node *i*. Then, betweenness centrality's values range from 0 to 1.

<u>*Network constraint*</u> [Burt, 1992 – page 57]: Burt (1992) argued that individuals that bridge a gap between other individuals or groups have an advantage because they

have access to unique information and resources. Those advantages dilute as the number of connections between the direct contacts of a node increase. The network constraint measure captures the degree to which a node *i* bridges disconnected individuals. The more interconnected the neighbors of node *i* are, the higher the node *i*'s constraint is, hence the lower the likelihood of accessing unique information.. This measure is similar in spirit to betweenness centrality. However, network constraint focuses on the structure of direct ties and the two-hop ties to a particular node. More importantly, unlike betweenness centrality, network constraining suggests that there is a competitive element in having a good position in the network. The value of being a link between disconnected groups is significantly reduced if there are other people who also connect the groups. Network constraint is mathematically defined by the following formula:

$$NC(n_i) = \sum_j [e_{ij} + \sum_{q \neq i \neq j} e_{iq} e_{qj}]^2 \qquad \textbf{(Equation 8)}$$

Where e_{ij} represents the "energy" dedicated by node *i* to maintain the connection node *j*. The energy is typically assumed to be equally distributed across all connections of a node and it is computed as the reciprocal of the number of connections of that node. The minimum value of the network constraint measure is 0 and the maximum is a function of the number of neighbors a particular node has.

Traditional Factors Affecting Individual-level Development Performance: The software engineering literature emphasizes the role of cognitive and technical skills (Curtis, 1981). The work related to development time estimation models (see Kemerer, 1986 for a review) takes a more integrative approach relating the amount of time it would take to develop a particular piece of code to several classes of factors such as task and project characteristics as well as individual-level attributes. Finally, the work in the information systems literature (e.g. Rasch & Tosi, 1992) suggested additional socio-psychological factors (role ambiguity and goal attributes) that affect individual-level software development productivity.

In this study, I collected data on the developer's ability and skill using three variables: programming experience, domain experience and formal training in computer science or related field. I also collected attributes of the development tasks. *Programming Experience* was collected from archival data provided by the human resources department and it represented the number of years of programming experience the developer had prior to joining the company. I transformed the variable into a monthly measure and it was incremented on a monthly basis through out the time period covered by the study. *Domain Experience* was collected from archival data provided by the human resources department and it represented the number of years of developing software in the same domain prior to joining the company. As with the case of programming experience, I transformed the variable into a monthly measure and it was incremented on a monthly basis through out the time period covered by the study. All developers had at least a Bachelor level *Education* in computer science or a related field. This variable was measured in years and the following formula was used to account for graduate degrees: Education = 4 + 2 * MSc + 5 * PhD, where MSc and PhD are dichotomous variables indicating whether the developer completed a Masters (or equivalent) degree and a Doctoral level degree.

Some of the developers could work in more complex areas of the system, hence requiring more code to be developed as part of a modification request or a particular change to the software. I captured that variability by computing the *Average Change Size* in non-blank-non-comment lines of code for each developer's work based on the set of modification requests resolved in each month. I also considered the variable *Group*, which represents the formal team developers belong to, because the technical properties of the components developed by each team differ, potentially, the outcome variables considered in this study. These last two measures captures an important part of the "technical properties" of the development work that the productivity estimation models (e.g. Kemerer, 1986) argue are important factors to consider.

Unfortunately, I was not able to collect any of the socio-psychological factors mentioned in the information systems literature. However, considering the *Group* measure as a random-effect in the multi-level model, described later in the chapter, should address some of the variability that might exists across teams in terms of definitions of goals and the ambiguity of the roles that each developer have in each specific team.

Other Control Variables: Since the network data is based on actual interactions through a communication means (IRC), the developer's propensity to use that communication tool is a potential factor that affects the individual's ability to coordinate his or her work and ultimately the individual's performance. Hence, a control variable *CommUsage* that captures the number of conversations the developer participated in IRC across all channels and across all topics during a particular time period was added. The tool PieSpy (Mutton, 2004) was used to construct the relational data from the IRC logs. PieSpy uses the temporal density approach to detect non-directly addressed interactions. This variable differs significantly from the network measures on the coordination networks because *CommUsage* also accounts for any other interactions related to modification requests not included in the dataset or any other non-work related communication. A similar approach was used to compute the CommUsage measure from the MR tracking system data. In this case, I also included all the modification requests available from project A and I also considered the communication carried out through the CC-list as a proxy for propensity to use the tool. Finally, the *Time* variable indicates how many months have passed since the starting point of the analysis and captures variability on performance related to the passage of time. The values of this variable range from month $j = 0$ corresponding to November 2001 to month $j = 38$ corresponding to February 2005. It is important to highlight that a measure of familiarity with the system under development, *Tenure*, was also computed. However, it was highly correlated with the *Time* measure, so I did not include it in the analysis.

Results

General Patterns of Coordination Behavior

Using the interaction data from IRC and the MR-tracking system, I constructed monthly coordination networks. Figure 16 shows months 10, 20 and 30 from the MR-tracking system data. The general pattern of the coordination networks is a core-periphery structure (Borgatti & Everett, 1999) suggesting that a particular group of developers are at the center of the coordination activities and the exchange of information among engineers. The rest of the developers seem to rely solely on interactions with the centrally positioned developers for coordinating their tasks. Our IRC coordination data showed the same core-periphery pattern (see figure 17). The strong core-periphery patterns were analytically confirmed by using Borgatti and Everett's methods for fitting network patterns to a core-periphery structure. The average fit, based on the continuous model, across all 39 months was 0.721 with a minimum fit of 0.568 and a maximum one of 0.858. These results confirm coordination networks feature a relatively small number of people who play a special role as communication hubs.

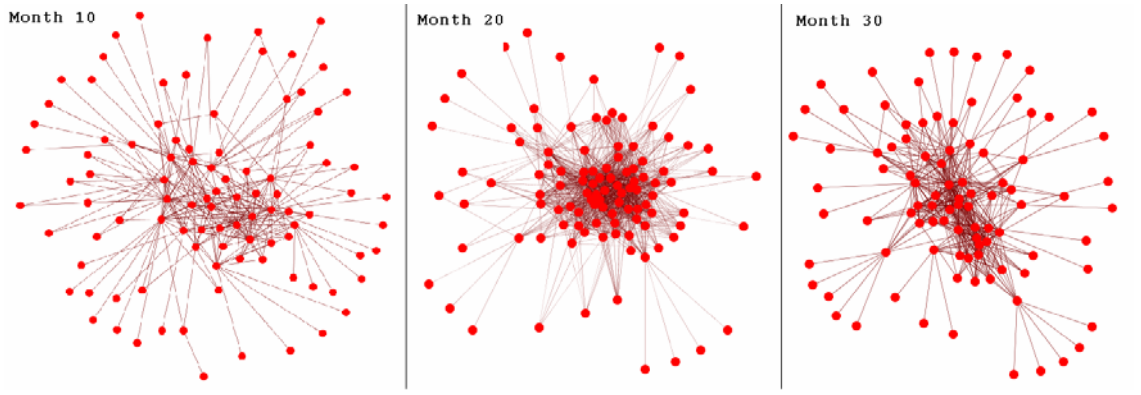

Figure 16: Over Time Coordination Patterns from the MR system data

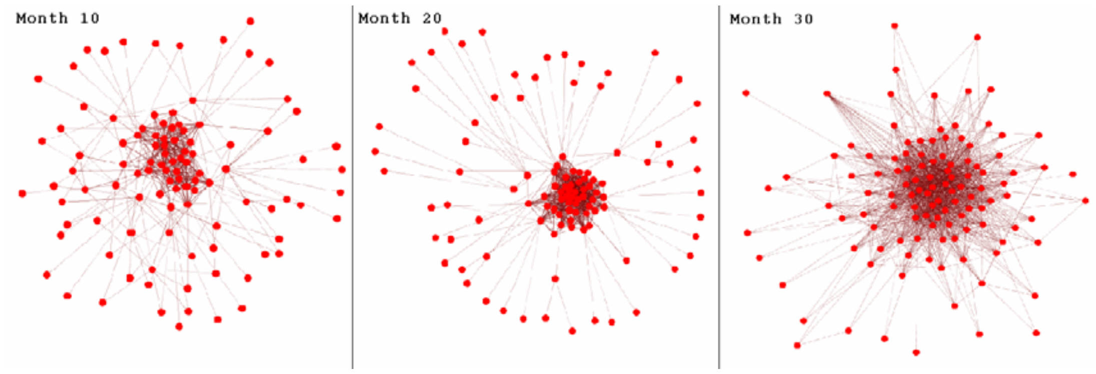

Figure 17: Over Time Coordination Patterns from the IRC data

In the next step of the analysis, the structural position of the developers in the coordination network was related to the developers' membership to formal teams and geographical locations. Figure 18 shows the coordination network from month 17 from the IRC coordination data where the developers are color-coded for membership to formal teams (left hand-side picture) and based on geographical location (right hand-side picture).

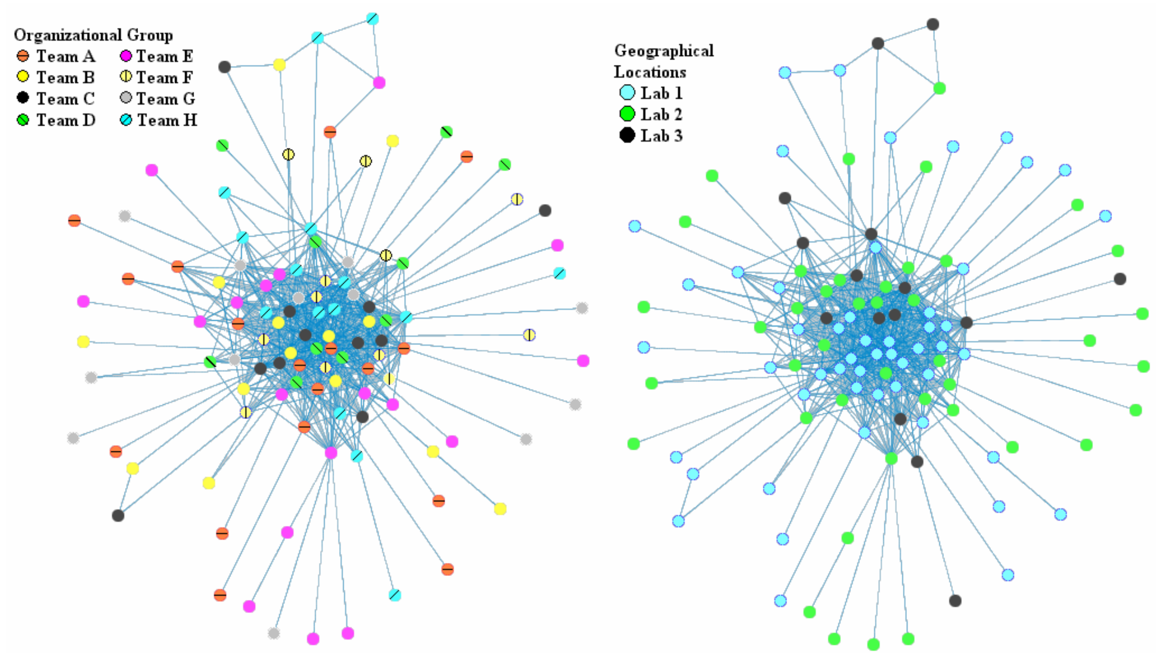

Figure 18: Coordination Patterns across Formal Teams and Geographical Locations

Figure 18 shows developers from all eight teams are represented in the highly interconnected core of the coordination network. In addition, a large portion of the developers are in the periphery and most of the communication and coordination involves developers in the core. Figure 18 also shows developers in the core seem to act as gateways or gatekeepers to other teams and other geographical locations for the developers in the periphery as suggested in figures 16 and 17. The existence of gatekeepers replicates the findings Allen (1977) encountered in R&D organizations, and extends them to geographically distributed teams.

The role of the core group in terms of coordination across geographical locations was statistically examined using an ANOVA analysis to evaluate the frequency of interaction in a 2 x 3 factorial design where dyads were classified along two dimensions: same geographical location (yes or no) and position in the coordination network (both nodes in the core, both nodes in the periphery or a node from each group). I used the MR and IRC coordination data aggregated at the level of product release. Since the observations (the dyads) are not independent, I assessed the ANOVA results using a random replication procedure. I used 1000 and 5000 replications and all ANOVA results were consistent. I found statistically significant effects of geographical location (*F*=74.70, p<0.001), position in the network (*F*=93.95, p<0.001) and the interaction term on the frequency of communication (*F*=15.51, p<0.001). In the first release of the product, for instance, dyads within the same location (*mean*=137.43, *sd*=119.31) were more frequent than those across geographical locations (*mean*=67.84, *sd*=24.60). Those individuals in the core (*mean*=127.81, *sd*=98.65) communicated more frequently than those dyads in the periphery (*mean*=53.17, *sd*=23.87) or those dyads had one node in the periphery and one in the core (*mean*=93.08, *sd*=68.03). Considering the dyads where the individuals are not in the same geographical location, more frequent communication occurs when dyads have both developers in the core (*mean*=117.23, *sd*=80.96) relative to the cases where both individuals are in the periphery (*mean*=3.56, *sd*=2.97) or one developer is in the periphery and the other in the core (*mean*=47.95, *sd*=34.21). On the other hand, if the individuals in the dyads are in the same geographical location the mean frequency of communication is significantly higher. When dyads have both developers in the core, the mean frequency of interaction is 179.91 (*sd*=98.17), while in the case where both individuals are in the periphery the mean frequency of interaction is 75.06 (*sd*=42.53). Finally, if one developer is in the periphery and the other in the core the mean frequency of interaction is137.41 (*sd*=69.47). Figure 19 depicts the frequency of interaction residuals after correcting for the location and the network position effects as well as for the grand mean. Then, the interaction effect becomes clear, indicating developers in the core handle more of the coordination activity that crosses the geographical boundaries.

The analysis showed consistent results across all four releases of the product. I also replicated the ANOVA analysis in a random sample of 10 months to verify that aggregating the data at the release level was not influencing the findings. The results were consistent with those from the release-level analyses. In sum, the analysis suggests developers in the core carry more of the load of handling the communication and coordination across sites. The next step of the analysis examines whether this “bridging” role comes at the cost of reducing the direct contribution to the development effort.

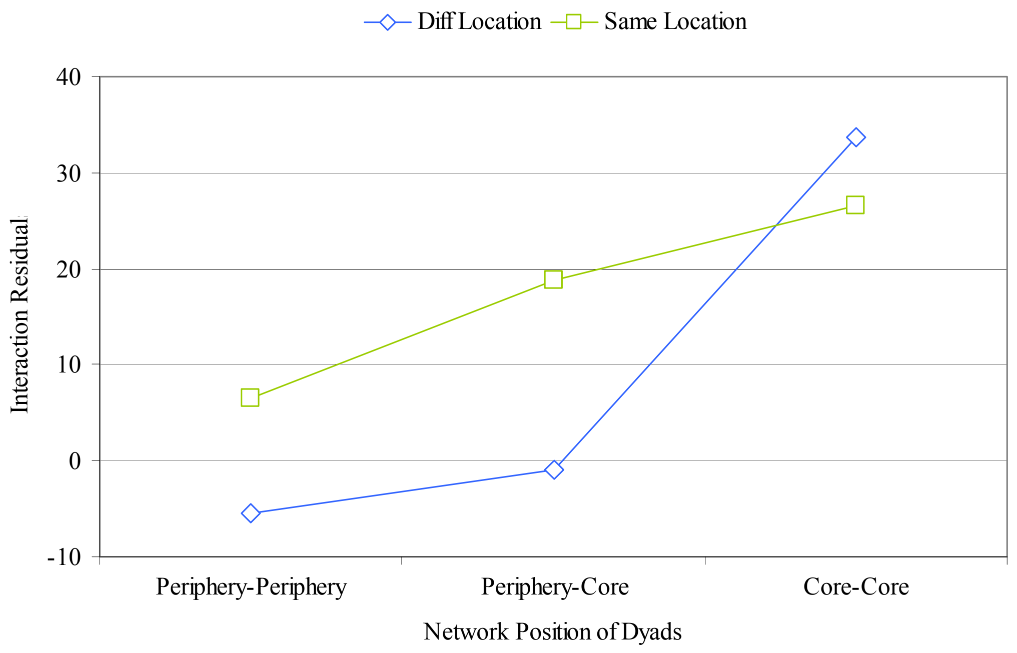

Figure 19: Location X Network Position Interaction Effect

On the Relationship between Network Position and Productivity

I examined the question of whether the most productive developers are part of the core group of the coordination networks both qualitatively and quantitatively.

Qualitative Analysis

In order to gain a better understanding of the composition of the core group in the coordination networks, I related membership to the core group to the developers' contribution to the development effort. For each month, the developers were ranked in terms of the amount of code contributed to the project and I divided the ranking into five groups: "highest" performers to "lowest" performers. Figure 20 shows an example of a coordination network (month 17) where each developer is categorized into a productivity group. The graph suggests that the majority of the developers in the core are high performers (cyan nodes), while less performing developers tend to remain in the periphery of the coordination network. However there are several interesting cases. There are several high performing engineers that are in the periphery (yellow nodes) and they seem to coordinate their work minimally. On the other hand, there are low performing individuals positioned very centrally in the coordination network (three green nodes in the core).

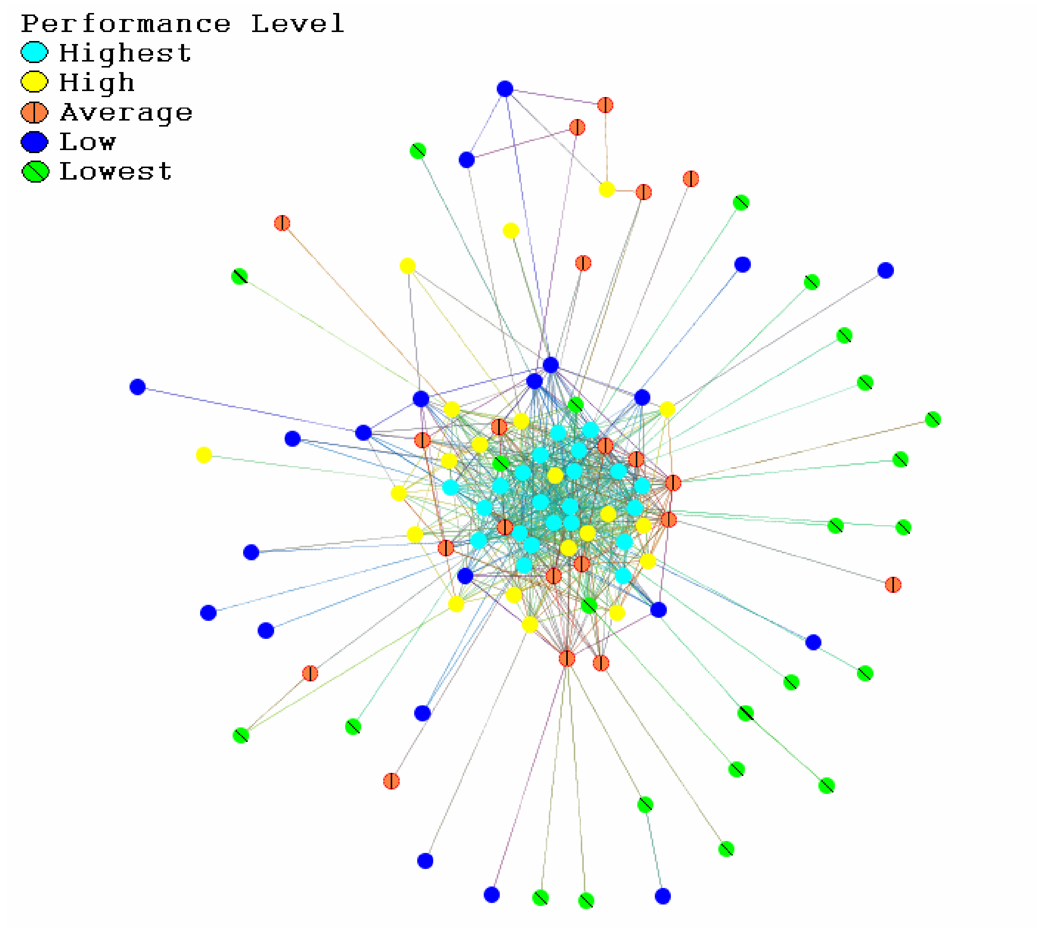

Figure 20: Coordination Patterns and Productivity

In order to evaluate if the pattern suggested by figure 20 persisted over time, additional analyses were performed. First, I compared the monthly coordination networks along two dimensions: how many developers were in the core of the coordination network in each month, and how many top performing developers were part of the core in each month. Borgatti & Everett's (1999) method to identify the core group in each monthly network was used. Figure 21 shows the number of developers in the core group averaged 30, with a minimum number of 14 and maximum of 42 engineers. In addition, the number of engineers from the highest productivity group ranged from 15 to 21 over the 39 months of data.

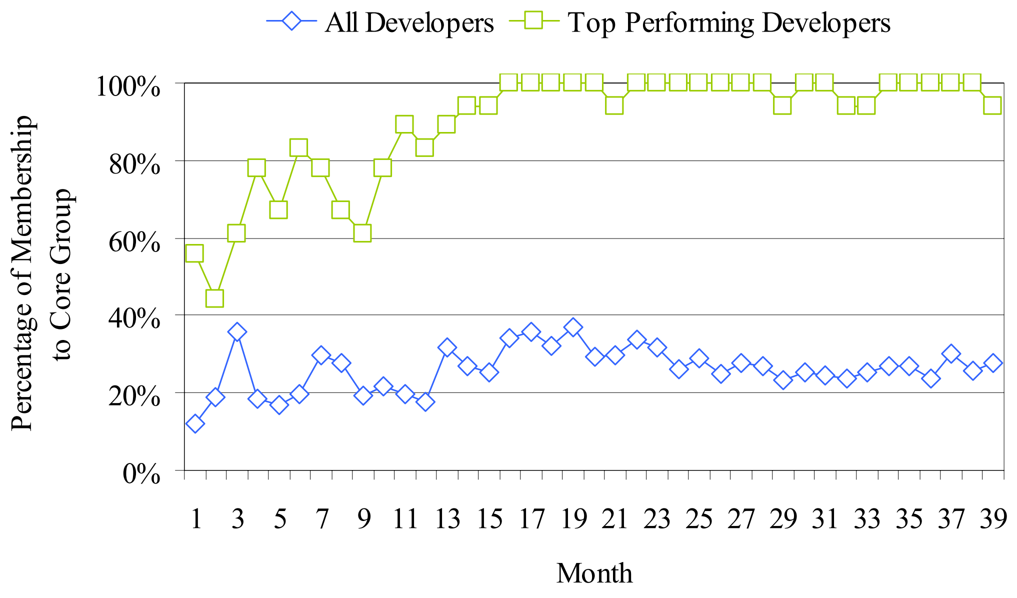

Figure 21: The Size of the Core Group over Time and Top Performers Membership

In addition, during the first 1/3 of the time covered by the data, the composition of the core group varied significantly. However, after month 15 most of the top performers consistently belong to the core on the coordination network. Interestingly, there are several instances where low productivity developers are also part of the core coordination group (see Figure 22). An examination of the characteristics of the development tasks performed by the engineers suggested significant differences across productivity groups in terms of the average number of source code files affected by modification requests. In several months (e.g. 3, 8, 17, 27, and 38), developers in the lowest two productivity groups worked on modification requests that affected, on average, the highest number of source code files. The examination of the modification requests and changes to the source code indicated that those developers tended to focus on developing features of the system that cut across numerous subsystems such as tracing and security functionalities. Then, modifications to those files would require coordinating work across several groups of individuals. In fact, it is that need to interact with many other engineers that seems to drive some of the lower performing developers to the core of the coordination network as figure 22 shows.

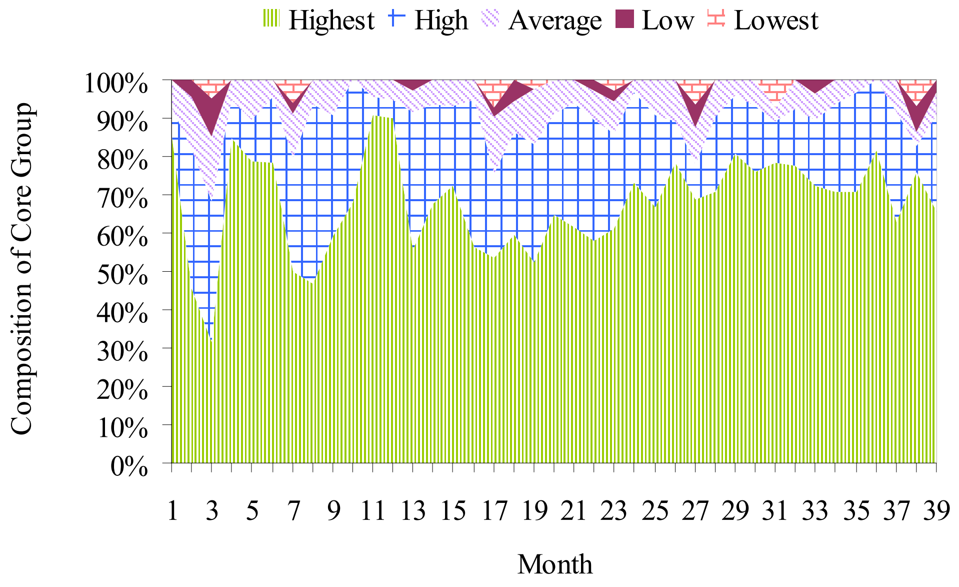

Figure 22: Composition of the Core Group over Time by Productivity Levels

Quantitative Analysis

The longitudinal dataset used in this study has characteristics that render traditional linear regression models inadequate for statistical analysis. As is the case with any longitudinal dataset, the autocorrelation between the observations of the same measures over time will violate the independence assumptions of a traditional linear model. Hence, a multi-level model (Singer & Willet, 2003), also known in the literature as mixed models, was used to examine the effect of coordination behavior on individual-level performance and its evolution over time.

The multi-level modeling approach allows variation at several levels within the model. The specification of a multi-level model includes fixed effects and random effects that may be applied to multiple variables for a given stream of longitudinal data. For instance, the impact of time may vary across individuals – a multi-level model allows an analysis that accounts for this type of variability. In this dataset, I have a stream of data for each developer and the model allows variation of both the intercept, the influence of time and the impact of working in a particular development team on the productivity of the individual. In this way, I account for the effects of individual-level factors (e.g. domain experience), characteristics of the development work that are specific to a development group as well as seasonal and other time-related variability in our population.

Formally and based on Singer and Willet's (2003) notation, the statistical model is described by equations 1 and 2. First, level-one model (equation 1) is created, which represents a basic linear regression for each individual. In this level-one model, there are *j* observations over time for each individual *i*. The variable $Time_j$ indicates how many months have passed since the starting point of the analysis. The variable $Group_j$ indicates the formal team the developer *i* belongs to. $TraditionalMeasures_j$ refer to the set of traditional factors that affect individual level performance described in the previous section, and $NetworkMeasures_j$ represent the set of network measures also described in the previous sections. It is important to clarify that this is a descriptive equation and it does not imply that all 6 network measures are included in the analysis. As discussed later in the results section, the correlation among the independent variables drives the selection and inclusion of specific variables into each statistical model.

$$\begin{aligned} Y_{ij} = \pi_{0i} + \pi_{1i}(Time_j) + \\ \pi_2(NetworkMeasures_j) + \\ \pi_3(TraditionalMeasures_j) + e_{ij} \end{aligned} \quad \textbf{(Equation 9)}$$

The coefficients of the model that are specific to each individual can then be further expanded for additional insight by defining the level-two model. At this level, variations at each time period are allowed. Inserting equation 10 into equation 9 results in the basic model used in the analysis.

$$\begin{aligned} \pi_{0i} = \gamma_{00} + \gamma_{01}(Time_j) + \gamma_{02}(Group_j) + \zeta_{0i} \\ \pi_{1i} = \gamma_{10} + \gamma_{11}(Time_j) + \gamma_{12}(Group_j) + \zeta_{1i} \end{aligned} \quad \textbf{(Equation 10)}$$

In this multi-level model, I assumed the effects of the network measures (π_2) and traditional variables (π_3) were constant across all individuals. On the other hand, the effects of time and group membership were allowed to fluctuate across developers.

The normality assumptions in the multi-level model described in the previous paragraphs are similar to the assumptions required by traditional linear regression models. In order to satisfy those assumptions it was necessary to perform a log-transformation in the dependent variables, *NumChanges* and *NumMRs*, as well as some of the independent variables as indicated in table 22. It is important to highlight that the statistics presented in table 22 are computed across all 39 months.

I evaluated the pair-wise correlations among all the independent variables to identify potential collinearity problems. Overall, the pair-wise correlations had acceptably low levels (below 0.20) with the exception of the pairs Programming Experience-Domain Experience, Degree and Eigenvector Centrality with all other network measures. These correlations were all statistically significant and with values higher than 0.45. Given these collinearity issues, I assessed the effect of each network measure separately using different multi-level statistical models that only have the traditional factors plus one network measure (see table 23). For each statistical model, I report the Akaike's Information Criteria (AIC) which is an indicator of the explanatory power of the model. Then, using a different model for each network measure allows us to compare the explanatory value of each network measure separately.

Table 22: Descriptive Statistics for IRC dataset

	Mean	SD	Skew	Kurt.	Min	Max
NumBugs (log)	0.518	0.911	0.083	2.139	0.000	4.189
NumChanges (log)	1.317	1.575	-0.592	2.976	0.000	5.298
Education (log)	1.626	0.288	0.873	2.174	1.386	2.397
Programming Exp. (log)	4.158	1.002	-0.264	1.901	2.484	6.095
Domain Exp. (log)	3.306	0.801	0.163	2.808	2.484	5.318
Avg. Change Size (log)	2.808	0.976	-0.128	4.144	0.000	6.873
Comm. Usage	23.758	35.159	1.364	3.697	2.000	165.000
Degree Centrality	0.082	0.122	1.228	3.076	0.000	0.473
Betweenness Centrality	0.072	0.021	1.178	2.973	0.000	0.282
Eigenvector Centrality	0.078	0.059	1.113	3.117	0.000	0.991
Network Constraint	0.230	0.321	1.363	4.417	0.000	1.941
Local Clustering	0.354	0.418	0.405	1.297	0.000	1.000

Table 23 reports the results of the analysis using the dataset collected from communication and coordination activity using the online-chat tool (IRC). There are two baseline models that examined the effects of *Time*, *Education*, *Programming Experience*, *Domain Experience* and the *Average Size of Changes* for each dependent variable. The results indicate that over time, developers become more productive in terms of changes submitted as the statistically significant and positive coefficient on the variable *Time* indicates. However, *Time* does not have a statistically significant effect on the number of modification requests resolved. One possible explanation of this finding is that, as the system matures, modification requests become more difficult to resolve and they might involve more software changes as well. As expected, the larger the average size of the changes to the software developers submit, the lower the number of changes those developers submit and the lower the number of modification requests they resolve. As indicated earlier, *Programming Experience* and *Domain Experience* are highly correlated, as a result their effects was separately explored in Models 1, 2, 8 and 9. Consistent with prior research, higher levels of programming and domain experience increased the number of modifications requests resolved per developer and the number of changes submitted by the developers on a monthly basis. The value of the AIC of a model tells us the explanatory power of the model. In the case of comparing two models with the same number of predictors, the lower the AIC the more variance is explained by a particular model. In the case of models 1, 2, 8 and 9, *Domain Experience* has a higher explanatory value than *Programming Experience*. Given this difference, in the remaining models (3 through 7 and 10 through 14) of table 23, I included *Domain Experience* when evaluating the effects of the network measures.

In models 3 through 7 and 10 through 14 from table 23, the effects of position in the coordination network were explored. In models 3 through 6 and 10 through 13, I assessed the role of one network measure at a time so the explanatory power of each individual network measure could be examined. All network measures had a statistically significant and positive effect on individual-level performance. However, an examination the AIC values for each of the models, indicated that the impact of each measure varies substantially which suggests that specific structures of the social system of interactions are more beneficial than others. For instance, being highly connected (high *Degree Centrality* or high *Eigenvector Centrality*) has a higher impact on performance (based on a lower AIC of the model) relative to brokering information between disconnected groups (low *Network Constraint*). In the context of software development, these results suggest individuals benefit by ample access to information rather than by controlling the flow of information.

Table 23: Results of the Multi-level Regression Model using the IRC data

Effects on MRs Resolved							
	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	Model 7
Intercept	0.871*	0.473*	0.155*	0.407*	0.452*	0.570*	0.499*
Time	-0.003*	-0.004*	-0.003*	-0.004*	-0.004*	-0.004*	-0.004*
Education (log)	-0.021	-0.011	-0.016	-0.011	-0.011	-0.012	-0.011
Prog. Exp. (log)	-0.017						
Dom. Exp. (log)		0.057+	0.054+	0.058+	0.060+	0.056+	0.056+
Comm. Usage	0.015*	0.015*	0.014*	0.015*	0.016*	0.015*	0.015*
Chg. Size (log)	-0.10*	-0.09*	-0.054*	-0.089*	-0.097*	-0.111*	-0.101*
Degree Cent.			1.491*				
Between. Cent.				1.768*			1.733*
Eigen. Cent.					0.414*		
Network Const.						0.112*	0.106*
AIC	7578	7574	7361	7548	7566	7571	

Effects on Changes Committed							
	Model 8	Model 9	Model 10	Model 11	Model 12	Model 13	Model 14
Intercept	3.537*	3.311*	2.862*	3.147*	3.254*	3.186*	3.005*
Time	0.004*	0.006*	0.009*	0.006*	0.007*	0.007*	0.006*
Education (log)	-0.024	-0.016	-0.025	-0.015	-0.016	-0.014	-0.012
Prog. Exp. (log)	0.064+						
Dom. Exp. (log)		0.101*	0.086*	0.097*	0.106*	0.104*	0.100*
Comm. Usage	0.004*	0.004*	0.002*	0.004*	0.004*	0.004*	0.004*
Chg. Size (log)	-0.538*	-0.537*	-0.468*	-0.512*	-0.533*	-0.522*	-0.494*
Degree Cent.			2.383*				
Between. Cent.				5.101*			5.156*
Eigen. Cent.					1.290*		
Network Const.						0.140*	0.160*
AIC	7578	7537	6955	7293	7421	7528	7278

(+ p < 0.05, * p < 0.01)

On the other hand, the relatively high impact of *Betweennes Centrality* would suggest an opposite interpretation: brokering disconnected individuals or groups could be beneficial for a developer. However, the role of *Betweenness Centrality* can be explained by examining the differences in the scope of the structural properties captured by *Betweenness Centrality* and *Network Constraint*. The later two measures only consider the immediate connections of a particular node. In contrast *Betweenness Centrality* considers the entire network. Therefore, an individual could have high *Betweenness Centrality* if he/she is part of a highly interconnected sub-group within the network as it would be the case in a core/periphery type of structure. In this case, *Betweenness Centrality* and *Degree Centrality* would have similar effect.

I examined the relationship between *Betweenness Centrality* and *Network Constraint* in more detail. First, the pair-wise correlations between *Betweenness Centrality* and *Network Constraint* was positive and below 0.15. This is contrary to what is expected if high values *Betweenness Centrality* stemmed only from brokering disconnected individuals or groups. A more rigorous analysis is reported in models 8 and 14 (table 23). A comparison of the results from model 6 and 7 indicates that including *Betweenness Centrality* does not alter the sign and statistical significance of the estimated coefficient for *Network Constraint*. A similar observation can be made comparing model 13 versus model 14. Then, these results indicate that the effect of *Betweenness Centrality* is complementary to the effects of *Network Constraint.* This finding is consistent with the argument that *Betweenness Centrality* is similar to the effect of *Degree Centrality* as figures 16, 17, and 18 suggest. Hence, access to information is more beneficial than controlling the flow of information.

Finally, this analysis was also performed on the coordination data from the MR tracking system. Table 24 reports the results and, overall, they are all consistent with those reported in table 23. In the case of the MR data, all network measures were highly correlated, hence, the models containing multiple network measures were not feasible.

I also used data from project D to quantitatively examine the relationship between network position and individual-level productivity. In the case of project D, the coordination activity data was collected through a survey instrument (see appendix A). A self-assessed measure of productivity was also collected using the survey. Forty seven of the eighty three developers completed the survey resulting in a response rate of 56.62%. A second measure of productivity, number of modification requests resolved, was collected from software repositories corresponding to the two development iterations of project D under study. Unfortunately, data associated only with nineteen of the eighty three engineers was available.

Table 24: Results of the Multi-Level Regression Model using the MR data

Effects on MRs Resolved						
	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Intercept	0.871*	0.473*	0.393*	0.488*	0.475*	0.534*
Time	-0.006*	-0.004*	0.001*	-0.003*	-0.004*	-0.001*
Education (log)	-0.021	-0.011	-0.004	-0.010	-0.009	-0.008
Prog. Exp. (log)	-0.017					
Domain Exp. (log)		0.057+	0.087+	0.061+	0.058+	0.063+
Comm. Usage	0.026*	0.026*	0.048*	0.028*	0.027*	0.025*
Avg. Chg. Size (log)	-0.101*	-0.099*	-0.116*	-0.103*	-0.099*	-0.115*
Degree Centrality			4.481*			
Betweenness Cent.				2.381*		
Eigenvector Cent.					0.473*	
Network Constraint						0.504*
AIC	7577	7573	7262	7525	7572	7463

Effects on Changes Committed						
	Model 7	Model 8	Model 9	Model 10	Model 11	Model 12
Intercept	3.356*	3.310*	3.374*	3.314*	3.316*	3.307*
Time	0.004*	0.007*	0.004*	0.006*	0.007*	0.006*
Education (log)	-0.024	-0.016	-0.018	-0.017	-0.017	-0.016
Prog. Exp. (log)	0.064+					
Domain Exp. (log)		0.101*	0.083*	0.096*	0.098*	0.098*
Comm. Usage	0.006*	0.007*	0.002+	0.005*	0.007*	0.007*
Avg. Chg. Size (log)	-0.538*	-0.537*	-0.531*	-0.534*	-0.537*	-0.534*
Degree Centrality			1.767*			
Betweenness Cent.				1.870*		
Eigenvector Cent.					0.881*	
Network Constraint						0.114*
AIC	7541	7536	7491	7507	7524	7536

(+ $p < 0.05$, * $p < 0.01$)

Table 25: Results from Multi-Level Regression Model using Project D data

Effects on Self-assessed Productivity		
	Model I	Model II
Intercept	0.3092**	0.0730*
Degree Centrality	6.6629**	3.5074**
Network Constraint		4.7381**
N	43	43
AIC	137	71

Effects on MR resolved		
	Model III	Model IV
Intercept	0.0996	0.0462
Degree Centrality	4.3239**	3.6196*
Network Constraint		1.0711*
N	19	19
AIC	213	209

(* $p < 0.05$, ** $p < 0.01$)

Table 25 reports the results which are consistent with those reported in table 23 and table 24. Developers that are centrally positioned in the coordination network and interact with other highly interconnected engineers tend to be more productive.

Stability of Coordination Patterns

In this section, I examine the stability of the coordination patterns exhibited by the coordination networks using several approaches. First, I look at the overall change in the networks. Secondly, changes in the structural position of the developers are examined. Finally, I explore the relationship between coordination needs and coordination activities and how that relationship impacts the stability of the coordination patterns. Since a product release represents a clearly identifiable unit of analysis, I aggregated all the coordination activity at the level of a release of the product.

In the first step of the analysis, the equivalent of the Hamming distance between networks of each release of the product was computed. This approach identifies the linkages between developers that exist only in one of the two graphs, then, dividing that number over the total number of linkages possible gives the rate of change. Figure 23 shows the results and, overall, the rate of change in the coordination patterns is small in both communication means. For instance, the average change in the IRC coordination from release to release is about 3% which corresponds to 193 different dyadic connections or 1.7 connections per developer. The coordination activity over the MR tracking system has a higher rate of change than IRC coordination. However, in both cases, the rate of change decreases over time to relatively low levels.

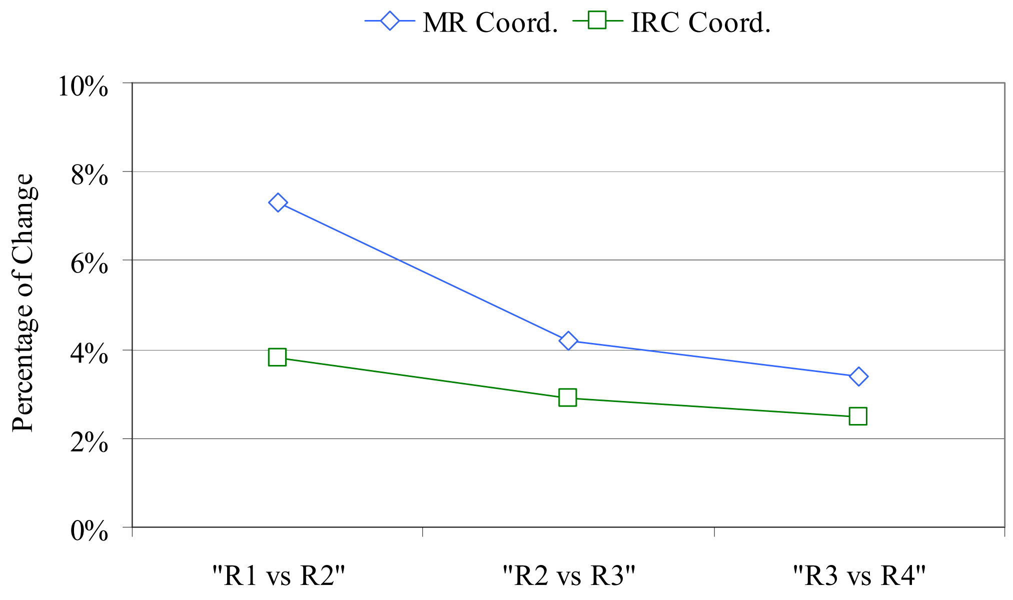

Figure 23: Amount of Change in Dyads Connections

The second approach to examine the stability of the coordination patterns consisted on comparing the ranking of the developers' degree centrality in the coordination network. For each release, the degree centrality measure of each developer was computed and a rank based that network measure was constructed. Then, I compared the ordered sets of developers using Kendall's *tau* which represents the probability that the rankings are the same. Table 25 reports the results of the analysis. The rankings in the MR-based coordination network are very stable, particularly in the last three releases where the probability of developers exhibiting the same pattern of coordination activities was above 0.78. The stability of the IRC coordination pattern is lower than those of the MR coordination patterns, but it is still quite significant

Table 26: Stability of the Coordination Networks

	MR Coordination	IRC Coordination
Release 1 vs. Release 2	0.6704**	0.5307**
Release 2 vs. Release 3	0.8092**	0.6284**
Release 3 vs. Release 4	0.7863**	0.5541**

(* $p < 0.05$, ** $p < 0.01$)

Drivers of Coordination Patterns

Finally, I examined the relationship between coordination requirements of a particular release and the corresponding coordination activities for that release. The coordination needs were computed using the FCT method described in chapter 4. Considering the coordination networks and the coordination requirements in matrix form, I used Matrix Regression QAP (Krackhardt, 1988) to examine the relationship between them. MRQAP uses the dyadic relationship information contained in a set of matrices as independent variables to predict the linkages on a matrix considered as the dependent variable. In the analysis, I used the coordination requirements from a particular release to predict the coordination activity in that release. I also examined the impact of the coordination activity and coordination requirements from a previous release. For instance, for the 4th release of the product, I examined how the coordination activity of release 4 was impacted by the coordination requirements from releases 4 and 3 as well as by the coordination activity from release 3. Table 26 shows the standardized coefficients and the R^2 from the MRQAP analysis.

Table 27: Predicting Coordination Activities

MR-based Coordination				
	Rel. 1	Rel. 2	Rel. 3	Rel. 4
Previous Release Coord. Activities	--	0.47*	0.75*	0.66*
Previous Release Coord. Needs	--	0.09*	0.06*	0.07*
Current Release Coord. Needs	0.47*	0.21*	0.38*	0.44*
R^2	0.162	0.567	0.621	0.579

IRC-based Coordination				
	Rel. 1	Rel. 2	Rel. 3	Rel. 4
Previous Release Coord. Activities	--	0.31*	0.53*	0.49*
Previous Release Coord. Needs	--	0.09*	0.14*	0.15*
Current Release Coord. Needs	0.37*	0.34*	0.37*	0.48*
R^2	0.228	0.629	0.747	0.714

(* $p < 0.01$)

Table 26 shows that coordination requirements from release N have a statistically significant effect on predicting coordination activity for release N. However, the most significant factor predicting coordination activity in release N is the coordination activity patterns from the previous release N-1. Given the results reported in the previous section in relation to the stability of the coordination networks, this particular finding is not surprising. The results suggest that once particular coordination and communication paths are established, they tend to persist over time even though there might be an explicit coordination need that would justify the existence of such information conduit.

Discussion

In this chapter, I presented a longitudinal analysis of coordination activities in a geographically distributed software development project. The results showed that developers that are positioned centrally in the social system of information exchanges and coordination perform a critical bridging activity across formal teams and geographical locations. These findings are consistent with past research highlighting the critical role "liaisons" individuals play in the performance of teams and projects (Ancona & Caldwell, 1992; Hauschildt & Schewe, 2000). However, the analysis also revealed those same individuals contributed the most to the development effort. More interestingly, in the research setting, the "liaisons" emerged over time from each development group, contrary to view typically discussed in the literature where these key roles are formally established (Ancona & Caldwell, 1992; Hauschildt & Schewe, 2000, Sangwan et al, 2006). Individuals in such formal roles, with different expectations and responsibilities from the rest of the engineers in a software development effort, might face important challenges stemming from the dynamic nature of technical and task dependencies. Future research should examine the differential impact, if any, of formal versus emergent "liaisons" roles.

The analysis also showed that the patterns of coordination were relatively stable. In fact, the stability of the coordination patterns increased over time on both communication means, MR and IRC, used by the development organization. These findings are consistent with past research indicating once established patterns of communication and coordination resist change (Henderson & Clark, 1990). Moreover, Henderson and Clark (1999) argued that the stability of the communication paths could be detrimental to the development organization because those communication conduits might not be the appropriate ones when the product structure changes. This line of research highlights the potential detrimental effects of the stability of patterns of communication and coordination.

However, past research has overlooked the potential positive impact of the structure of those communication and coordination patterns, particularly, in relation to their stability against organizational changes. Particular structures of coordination patterns might be more resilient to changes in the organization such as turnover. For instance, social network research has shown that communication patterns can be drastically affected by removing particular individuals from the network (Carley et al, 2001). However, those effects are contingent on the topological attributes of the network (Frantz et al, 2007). For instance, core-periphery networks tend to be more resilient than other structures (Frantz et al, 2007). In our research setting, there were negligible levels of turnover which could have benefited the stabilization of communication and coordination patterns, particularly across geographical locations. Further research is required to explore the relationship between the stability of communication and coordination patterns and turnover as well as changes in the architecture of software systems.

In addition, stable patterns of communication and coordination with particular structure might have benefits in relation to changes in the product structure. For instance, the core-periphery structure found in our research setting involved members of all development teams. One could argue that such structure could act as a "council" where relevant pieces of information from all parts of the system are shared and understood. Then, the changes in dependencies introduced by modifications in the product could be more easily identified. Recognizing the changes in dependencies might be significantly more challenging in other types of communication structures such as hierarchical because such tasks would reside in a group or individual not involved directly in the software development process.

CHAPTER 8: APPLICATIONS

Collaboration and communication tools are an integral part of the software development process (Sarma, 2005). Sarma (2005) argued the various types of tools could be grouped in terms of the role they play in the development process: communication, artifact management, and task management. Communication tools, such as email, instant messaging, and on-line chat systems (e.g. Internet Relay Chat), provide the basis for coordination of activities in geographically product development project (Karolak, 1998). Tools such as document management systems and software configuration management systems constitute the traditional artifact management tool. Finally, task management tools, such as Bugzilla or Manthis, provide the mechanisms to allocate tasks and monitor their progress. Researchers have claimed that using a combination of tools from all those three groups is considered a good practice that leads to higher quality software products (Halloran & Scherlis, 2002). However, those tools ought to be integrated appropriately with software development processes and organizational structures in order for GDSD organizations to experience the benefits of such tools (Cataldo et al, 2007).

In recent years, researchers have focused on providing development environments that facilitates communication, collaboration and coordination. One approach has been to integrate simple collaborative components into team-centric IDEs (Booch and Brown, 2003). An example of such approach is Jazz developed at IBM Research and originally proposed as a collaborative extension to Eclipse (Cheng et al, 2003). It later evolved into an independent application that combined the original team-centric concepts with a process-centric approach (Jazz, 2007). A related and complementary research approach has focused on assisting developers in identifying dependencies among parts of the software systems as well as dependencies among development tasks. Tools such as TUKAN (Schummer et al, 2003), Palantir (Sarna et al, 2003) and Ariadne (Trainer et al, 2005) provide visualization and awareness mechanisms to aid developers identify and handle modifications to the same software artifacts such as source code files. In addition, Ariadne (Trainer et al, 2005) uses syntactic dependencies graphs to relate technical to social dependencies.

Unfortunately, existing IDEs or collaborative tools, such as those described in the previous paragraphs, do not help on identifying fine-grained implicit product dependencies as well as their related work dependencies. The results reported in this dissertation showed that failing to identify those types of dependencies are critical factors negatively affecting development productivity and software quality. The framework proposed in chapter 2 provides a mechanism that facilitates the identification of fine-grain work dependencies. Moreover, existing tools focused on software developers, however, the mechanism and measures proposed in this dissertation are also valuable for managers and other stakeholders such as project leads. The rest of this chapter discusses how the approaches to measure product and work dependencies used in this dissertation can be used to extend existing collaborative and coordination tools or implement new ones. The discussion is organized around the target users of those potential tools: software developers and stakeholders with managerial responsibilities in software projects.

Applications for software developers

The studies reported in this dissertation showed that coordination behavior on the part of software engineers that is congruent with coordination needs improves the resolution time of modification requests. In addition, the results indicated that explicit product dependencies such as syntactic relationships as well as implicit product dependencies (e.g. semantic or logical relationships) have an important and complementary effect on the failure proneness of software systems. In the following paragraphs, I describe how the congruence framework proposed in this dissertation could be utilized as part of collaborative and coordination tools to assist software developers.

Enhancing coordination needs awareness

Collaboration and task awareness tools are a natural application for the coordination requirements measure proposed in chapter 2. Part of the research effort of the CSCW community has been on improving traditional tools, such as email and instant messaging, which have become an integral part of work in the vast majority of organizations (Belotti et al, 2003; Wattenberg et al, 2005). The coordination requirements measure provides a way of identifying the email exchanges that are more relevant given the task interdependencies among individuals. This information would enable tools to provide an enhanced task management experience by, for instance, prioritizing to-do-lists and generating reminders to respond to task-specific emails based on the coordination requirements. This email sorting approach could be thought as a task-specific alternative to other social-based sorting techniques such as the one proposed by Fisher and colleagues (2006). A more recent set of tools, such as sidebars (Cadiz et al, 2002) and productivity assistants (Geyer et al, 2007), would also benefit from the congruence framework. These types of tools focus on activity-centric collaboration and, as argued by Geyer and colleagues (2007), the majority of the tools assume user intervention in terms of deciding what type of information to make part of the sidebar. The congruence framework would provide an automatic mechanism to identify people of interest giving a particular set of task dependencies among the workers.

In the context of software development projects, particularly those that involve tens or hundreds of engineers, identifying the right person to interact with and coordinate interdependent activities might not be a straightforward task. In fact, it is well established that software developers have serious difficulties identifying the right set of individuals to coordinate with (de Souza et al, 2004; de Souza et al, 2007; Grinter et al, 1999). The coordination requirement measure provides a mechanism to augment awareness tools that provide real-time information regarding the likely set of workers that a particular individual might need to communicate with. For instance, integrated development environments, such as Eclipse or Jazz, could use the coordination requirement information to recommend a dynamic "coordination buddy list" every time particular parts of the software are modified. In this way, the developer becomes aware of the set of engineers that modified parts of the system that are interdependent with the one the developer is working on. The concept of the "buddy list" in communication and collaboration tools is not a new idea. However, the novel contribution is to construct the "buddy list" from accurate estimates of the set of individuals more likely to be relevant to a particular developer in relations to the work dependencies, information which is captured by the coordination requirements measure.

Enhancing awareness of product dependencies

The software engineering literature suggests several types of technical dependencies in software systems. The earliest form of software dependencies (see, e.g., Stevens, Myers and Constantine, 1974) are syntactic relationships among modules of a system that are reflected in the software code by the use of functions, methods, variables and other programming language constructs. In more recent work in the software evolution literature, Gall and colleagues (1998) showed it is possible to uncover logical dependencies among modules that are not explicitly identified by traditional syntactic approaches. These two approaches correspond to the CGRAPH and FCT methods, respectively, discussed in chapter 4. The results reported in this dissertation showed that logical dependencies, such as those identified using the FCT method, are able to capture the dynamic nature of the software development process and they provide a better way of determining work dependencies among developers relative to syntactic dependencies. In chapter 5, I reported the implications of logical dependencies in the context of development productivity. In addition, chapter 6 reported in relevance of logical dependencies in relation to the failure proneness of a software system. These findings suggest that providing information (e.g. through visualizations) about product dependencies, particularly logical relationships, could have important benefits in terms of coordination and awareness among developers, and consequently, impact productivity and quality.

Tools such as TUKAN (Schummer & Haake, 2003), Palantir (Sarma et al, 2003) and Ariadne (Trainer et al, 2005; de Souza et al, 2007) provide visualization and awareness mechanisms to aid developers coordinate their work. Those tools achieve their goal by monitoring concurrent access to software artifacts, such as source code files, and by identifying syntactic relationships among source code files. Then, information is visualized to assist the developers in resolving potential conflicts in their development tasks. Using the measures proposed in this dissertation, these tools could be enhanced along two dimensions. First, they could provide an additional view of product dependencies. Using an approach such as the FCT method to identify logical dependencies, these tools would be in a position to provide complementary product dependency information to the developers which, as suggested by this dissertation, could be more valuable in terms of raising awareness among developers about the potential impact of their changes in the software system. Secondly, these tools could also provide a more precise view of coordination needs among developers. These tools focused on artifacts shared or modified by multiple developers. The coordination requirements measure goes beyond the identifying such dependencies, allowing developers to identify those files that have dependencies among themselves when those dependencies are not explicitly determined.

Other applications of the congruence framework

The congruence framework presented in chapter 2 has been utilized in other applications in the context of software engineering. Minto and Murphy (2007) proposed a technique that recommends experts to developers based on the emergent team information provided by the coordination requirements matrix (see equation 1). The authors found their approach have a positive and significant effect on the quality of the expertise recommendation over traditional approaches such as “who modified the module last”.

The congruence framework has also been applied to requirements engineering to maintain awareness amongst engineers working on related software requirements. Kwan and colleagues (2006) proposed an approach to visualize dependencies among software requirements and communication patterns among the engineers involved in those requirements. The authors argued that the visualization method would provide the necessary mechanism to promote coordination and maintain awareness of changes in requirements; however, they did not empirically evaluate those claims.

Managerial applications

Collaboration and communication tools used in software development provide very valuable awareness information as well as various communication and coordination mechanisms to developers. However, the effectiveness of those tools depends, in most cases, on the developers’ willingness and ability to adjust their coordination behavior based on the information provided by the tools. Even if developers modify their coordination behaviors, organizational and social barriers could still limit or impede establishing a useful interaction with other individuals in the project. Then, it is also important to provide managers and other decision-makers with appropriate types of information in order to enable them to identify patterns of communication and coordination that might be detrimental to the success of software projects. The work presented in this dissertation is a step forward in that direction and the following paragraphs discuss managerial applications of the results from this dissertation.

Project-wide view of coordination patterns

The analyses reported in chapter 7 suggested an approach to collect coordination activity from existing data repositories, such as a workflow tool, and provide a project-view of patterns of coordination behavior. Coupling visualizations of project-wide coordination patterns with statistical analysis of such data could be a powerful tool for different stakeholders in geographically distributed software development projects. For instance, in the project studied in chapter 7, the analyses found that developers that are positioned centrally in the social system of information and coordination exchanges tend to perform better than those in the periphery. But more interestingly, those centrally positioned individuals played a key role in promoting communication across development teams and locations, ultimately increasing the likelihood of success in the project.

The identification of unusual patterns of communication and coordination could be critical to the success of a product development project. For instance, in a study of communication and coordination in a jet engine design project, Sosa et al. (2004), highlighting the difficulty of managing cross-boundary interdependencies, provides examples of interdependent teams that did not interact. The lack of appropriate communication resulted in difficulties at the time of integrating the various subsystems. A project-wide view of coordination facilitates the identification of unexpected patterns such as the (a) lack of communication and coordination amongst teams or locations, (b) levels of communication and coordination that could be considered excessively high or (c) the existence of indirect coordination patterns. When managers and decision-makers combine those types of information with an assessment of the coordination requirements that are expected in the project, they are in a position to intervene in the project by sponsoring change in the patterns of coordination, modifying the structure of the product to reduce certain coordination requirements or both.

Another useful approach for managers and other decision-makers to identify potential sources of problems or difficulties is to compare a historical assessment of general coordination patterns against milestones or other critical events in a project. The organizational behavior literature showed the importance of time on how work evolves within workgroup projects of varying length (Gersick, 1988; Gersick, 1989). The research established that even though groups had widely varying amounts of time for their projects and progressed at different rates, workgroups tended to select the midpoint as a heuristic milestone and use it as a triggering mechanism to help ensure they will move fast enough to finish by their deadlines. A related finding was reported by Bass and colleagues (2007) in the context of global software development. The authors found that defects were given higher levels of attention the closer the reporting date was to the end of the development iteration. On the other hand, defects reported early in the development iteration tended to remain unresolved for extended periods of time, in several cases beyond the development iteration in which they were reported. Then, relating past patterns of coordination to relevant points in time in a development project could provide managers and other decision-makers with the information to understand where communication and coordination among developers or teams should be promoted in order to increase the quality of the product and the likelihood of success in the project.

Identifying critical software and organizational agents and units

The identification of key individuals in a system of social relationships has received significant attention from the sociology and social networks communities. Examples are the work on actor centrality that focuses on understanding the structural relevance of individuals in networks (e.g. Freeman, 1979; Bonacich, 1987), the work on identifying core and periphery structures formalized by Borgatti and Everett (1999) and the work on measurement of social capital (e.g. Burt, 1992). Borgatti (2006) extended the traditional concept of "key players" to highlight two distinctive perspectives: one focuses on the key player as maintaining the cohesiveness in the system of relationships and the second one focuses on the key player as connected to and embedded in the social network. The analyses reported in this dissertation apply two both perspectives. The results in chapter 7 showed the developers' structural position in the coordination network relates to the developers' contributions to the project as well as their role in bridging organizational teams and geographical locations. In fact, the analysis revealed that approximately 20% of the developers acted as bridges between formal teams and geographical locations and, simultaneously, those same individuals contributed an average of 57% of the implementation of the software system. Then, those results have several important managerial implications in term of (a) understanding who those individuals are in order to maintain those individuals motivated to continue to contribute and perform their roles are liaisons, (b) understanding the impact of developer turnover or organizational mobility in the flow of information exchange and coordination, and (c) identifying other developers that could be good candidates to perform those bridging functions. Tools could build on the qualitative and quantitative analysis presented in chapter 7 that identified key individuals groups using core-periphery models and traditional centrality measures. Moreover, tools could use the same type of data, such as coordination carried over a defect-tracking system, and apply other network measures to identify key individuals such cognitive demand (Carley et al, 2003) or the measures proposed by Borgatti (2006). This collection of network measures would provide valuable information regarding the three managerial implications outlined earlier in this paragraph.

The idea of "key players" could also be applied in the context of software artifacts such as source code files. The study of the impact of product dependencies on the failure proneness of software systems used the concept of node centrality in the set of relationships among source code files. The results showed that files with higher number of syntactic and logical dependencies were more prone to failure. However, the results also indicated that those source code files that had logical dependencies with other highly interconnected files were less likely to exhibit customer-reported defects. Then, these findings suggest an extension to the idea of modularity of a system to the level of "clusters" of source code files. Combining that relational information between software artifacts with pieces of information with developers' relational data, such as coordination patterns, a manager or other stakeholders would be able to better understand numerous critical aspects associated with the evolution of development projects such as the implications of task or role assignments among developers as well as the implications of major modifications to specific parts of the system.

CHAPTER 9: CONCLUSIONS

The identification and management of software dependencies is a fundamental problem in software development, particularly when development organizations are geographically distributed. This dissertation argued that modularization, the traditional approach used to reduce technical dependencies, is not a sufficient representation of work dependencies in the context of software development mainly for three reasons. First, as the results in chapter 5 show, the product structure-task structure relationship is not as simple as theorized. Second, the modularization techniques used in software development only consider one type of technical dependencies, syntactic relationships (Garcia et al, 2007). Those techniques disregard the technical relationships, such as logical dependencies, most relevant in determining work dependencies in software development. Thirdly, dynamic nature of the software development activities is better captured by logical product dependencies, as discussed in chapters 5 and 6. Hence, a new way of thinking about work dependencies in software development is needed. I proposed a method for measuring socio-technical congruence defined as the relationship between the structure of work dependencies and the coordination patterns of the organization doing the technical work. Two empirical studies assessed the impact of socio-technical congruence on development productivity and product quality. In addition, I explored how developers in a geographically distributed software development organization evolve their coordination patterns to overcome the limitations of the modular design approach.

The results indicated that higher levels of congruence were associated with lower levels of customer defects. However, the more product dependencies a module had with other parts of the system and the higher the amount of coordination requirements associated with each developer were found to be detrimental to the quality of the system. Higher levels of congruence were also associated with higher levels of development productivity. Moreover, the most productive developers exhibited two distinct characteristics: they coordinated their work more congruently than less productive developers and they played a critical role in coordination across teams and geographical locations. Collectively, these results have important implications for the design of collaborative tools as well as for organizing GDSD teams.

The rest of this chapter discuss the contributions and limitation of the work reported in this dissertation. I also present several research questions that the results from this dissertation suggest as promising areas for future work.

Contributions

This dissertation has important theoretical and empirical contributions to the software engineering, CSCW and organizational literatures. In terms of theoretical contributions, this dissertation presented a fine-grain view of coordination that addresses the limitations of traditional approaches from the organizational theory literature. The proposed framework for measuring socio-technical congruence provides the necessary machinery to examine the consequences of coordination requirements that are not satisfied. In addition, the congruence framework provides the sufficient flexibility to consider multiple types of product dependencies and their implications on the work dependencies encountered by product development organizations.

This dissertation also has significant empirical contributions. First, the empirical evaluation of the congruence framework showed the importance of understanding the dynamic nature of software development. Identifying the “right” set of product dependencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution time of modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively stable view of product dependencies that is not representative of the dynamism in product dependencies that emerges as software systems are implemented. On the other hand, logical dependencies provide a more accurate representation of the most relevant product dependencies in software development projects. The statistical analyses showed that when developers’ coordination patterns are congruent with their coordination needs, the resolution time of modification requests was, on average, reduced by 32% when considering the collective effect of all four measures of congruence. Generalizing, the empirical examination of the congruence framework and coordination patterns showed the tight relationship between team design, coordination and performance providing an important contribution to the organizational literature.

Secondly, this dissertation moves forward our understanding of the relationship between product and work dependencies and software quality. The empirical study reported in chapter 6 showed that logical dependencies among software modules and work dependencies are two of the most relevant factors affecting the failure proneness of software modules. For instance, the statistical analyses indicated that a unit increase in logical dependencies increased twice as much the likely of failure relative to the impact of syntactic dependencies. In addition, the proposed statistical models that included the different types of technical and work dependencies exhibit significantly better predictive power than recent models (e.g. Nagappan & Ball, 2007) that consider traditional factors such syntactic dependencies and churn metrics.

Finally, I presented a longitudinal analysis of coordination activities in a geographically distributed software development project. The results showed that approximately 20% of the developers were positioned centrally in the social system of information exchanges and coordination activities performing a critical bridging function across formal teams and geographical locations. In addition, those same individuals contributed the between 50% and 65% to the development effort in terms of implementing the software system in each released covered by the data. The analysis also revealed that the patterns of coordination become stable over time, and those patterns were only partially driven by the coordination requirements of the development tasks.

Limitations

It is also important to highlight some of the limitations of the work reported in this dissertation. First, the measures proposed as part of the congruence framework are contingent on assumptions about the software development processes used in the development organization as well as usage patterns of tools that assist the development effort such as defect tracking and version control systems. One key assumption is the possibility to identify (1) the set of source code files that were changed as part of a modification request and (2) the developers that made those changes. For instance, a policy of source code file ownership by particular developers could potentially bias the congruence measures. Developers that own a particular source code might appear as participants in the development effort associated with a modification request, however, that might not be the case. In other cases, such as open source projects (e.g. project B), the nature of the work in certain project is such that the information about which files changed together as part of a modification request is almost impossible to reconstruct in a reliable way.

The alternative approach of computing coordination requirements based on syntactic relationships also has its limitations. The method relies on tools that can reliably extract the dependency information among software modules for a specific programming language. More importantly, projects that use multiple programming languages will represent a challenge, particularly, in terms of determining syntactic dependencies that involve modules written in different programming languages.

Another limitation of the work presented in this dissertation is a potential concern for external validity of some of the empirical analyses. For instance, the study reported in chapter 5 examined only one system with particular properties that might be conducive to support the results found by the analysis. However, the processes and tools used by the development organization are commonplace in the software industry. Moreover, the general technical characteristics of the system are similar to other types of distributed systems developed into products in the software industry. Hence, I think the results are generalizable, particularly, in the context of development organizations responsible for delivering complex software systems.

Future Work

The work presented in this dissertation has also raised interesting questions to be addressed in future research work and the following paragraphs discuss them in detail.

Identification of coordination requirements in early stages of software projects

The empirical examination of the congruence framework showed the relevance of matching coordination activity with the fine-grained coordination needs that emerge in the development of software systems. However, the measure of congruence, as computed in the studies, relied on archival data to capture the appropriate product dependency information, the task assignment information as well as coordination activity carried out by the development organization. The promising results reported in this dissertation highlight the importance of identifying potential coordination needs as early as possible in the development process in order to provide the development organization with the appropriate communication and coordination mechanisms. Certainly such a task is a challenging one.

In early stages of a project, only architectural or high level design specifications of a system are available. Those documents by definition abstract a significant portion of the technical details of software systems in order to understand the overall attributes and relationships among the main components of a system. A higher level of abstraction could potentially hinder the identification of relevant technical dependencies and consequently, important coordination requirements. However, the use of standardized design and modeling languages, such as UML, might represent a way of overcoming these challenges. Researchers have proposed standard graphical representations of software architectures, called *views*, that capture different technical aspects of a software system (Clements et al, 2002). Examples of those graphical representations are the *module* view and the *components-and-connectors* view. Then, one approach is to construct a *coordination* view of the architectures that combines the product's technical dependencies with relationships among the organizational units responsible for carrying out the development work. In order to generate such representations, methods of identifying relevant dependencies from the technically focused views of the architecture are to be devised. One potentially promising approach is to synthesize the dependencies represented in the various types of UML diagrams (e.g. class diagrams, sequence diagrams, collaboration diagrams, etc) into a single set of technical relationships among modules. Such a method could be able to identify logical relationships (e.g. temporal relationships) among parts of the systems which, as shown in this dissertation, are an important factor driving the work dependencies in software development organizations.

The impact of formal roles in development organizations

The longitudinal analysis reported in chapter 7 showed developers positioned centrally in the social system of information exchanges and coordination activities performed a critical bridging activity across formal teams and geographical locations. The analysis also revealed those same individuals contributed the most to the development effort. More interestingly, and contrary to the views typically discussed in the literature (Ancona & Caldwell, 1992; Hauschildt & Schewe, 2000, Sangwan et al, 2006), the "liaisons" emerged over time from each development group.

In addition, these results challenge traditional thinking in the software engineering literature. As developers become more knowledgeable of the system and increase their productivity, they tend to be positioned in specific roles such as team leads. Those formal roles make the developers more visible to the overall organization, hence, it is expected that they would be involved in more communication, and coordination activities that facilitates the flow of information among teams. The additional responsibilities would negatively impact the individuals' direct contributions in the production of software code. However, the findings reported in this dissertation suggest an opposite situation where centrally positioned individuals in terms of communication and coordination are also highly productive individuals. Then, future work research is required to understand more closely the impact of formal roles on coordination in development organizations as well as the relationship between formal roles, coordination behavior and individual-level productivity.

Communication beyond team and location boundaries and individual-level performance

In addition to the issue of formal versus emergent roles, study III highlighted an interesting relationship between characteristics of the software development tasks and the developers' position in the coordination network. Although high-performers were more likely to be centrally positioned in the coordination networks, the longitudinal analysis showed that low-performing developers were also part of that core group at different points in time through the period covered by the data. An examination of the modification request reports revealed a particular set of developers worked on cross-cutting concerns such as logging, tracing and security. Those functionalities affected multiple parts of the system. The data suggested the developers implementing or modifying the cross-cutting concerns engaged in communication and coordination activities with several developers from other formal teams and geographical locations. Those findings raise several interesting questions. These developers had the opportunity to exchange information with developers working in different components of the system, then, do those information exchanges translate into an increase in the knowledge about the system and, consequently, higher development productivity? The data from project A examined in chapter 7 suggested that some improvements in productivity took place. The developers that worked on cross-cutting concerns tended to move up one or two categories in the productivity ranking after the months where they were part of the core group of the coordination network. However, the improvement in productivity did not translate into a consistent over time membership in the core group of the coordination networks. It is important to highlight that these findings are based on just five developers. Hence, more research is required to better understand the relationship between development tasks that promote interactions among engineers and the potential gains in development productivity. In addition, future research should examine if tasks, such as the implementation of cross-cutting concerns, are mechanisms that could promote the development of communication and coordination conduits among formal teams and development locations.

Applying the congruence framework in other types of tasks

This dissertation showed the congruence framework provided the appropriate machinery to measure the dynamic nature of work dependencies in software development and assess its impact on productivity and product quality. Although a natural progression of this work is to apply the congruence measures in other task settings, it is important to first discuss in more detail the general properties of task contexts where the usage of the congruence framework would be beneficial. The following paragraphs describe such properties.

Non-routineness: if all the steps required to performing a set of tasks can be identified a priori, then the nature of the potential interdependencies among those tasks is deterministic. Hence, the coordination mechanisms proposed in the organizational theory literature (e.g. Galbraith, 1973; March & Simon, 1958; Thompson, 1968) can be used. On the other hand, non-routine tasks are characterized by the impossibility to fully articulate and internalize all the necessary actions require to complete the task. Such a condition creates a dynamic set of task interdependencies. Hence, they constitute the ideal setting for the congruence framework.

Volatility of Coordination Needs: an issue related to the previous paragraph is the rate of change in the work dependencies associated with the non-routine tasks. The higher the volatility of coordination needs, the lower the applicability of the traditional coordination mechanisms. Then, the congruence framework would be better suited for a task context where dependencies constitute a dynamically evolving web of relationships.

Lack of global visibility: if a small group of individuals can harness a global understanding about the dependencies among of the tasks or the relationships among all the parts of a product under development, then that small set of individuals could be in a position were they can manage or facilitate the coordination among the relevant parties. The congruence framework would be useful in a setting (e.g. product development organizations that work with large and complex systems) were no individual, or small number of individuals, can have global understanding of the work or product dependencies. Then, the congruence framework could become the mechanism to manage the coordination complexity and provide assistance in the identification of coordination gaps.

The three properties described in the previous paragraphs represent the characteristics of an appropriate task context where applying the congruence framework would be valuable. However, once such a task context has been identified, an additional obstacle that could challenge the usage of the congruence framework is the availability of detailed task-related data which might not be as pervasive as in software development. Tools such as version control and defect tracking systems capture a wealth of information not typically available in other types of knowledge-intensive activities. There are promising technological developments in the area of delivering software applications that might help to overcome those problems. The concept of software as a service is growing in acceptance. In that model, applications are accessed as services (e.g. Google Docs or salesforce.com's customer relationship management tool). Such applications have the ability to capture a lot more information about the work performed by an interdependent group of individuals relative to the case where the applications are run separately on individual machines. Then, one could envision capturing information similar in nature as the one captured in software development projects by tools like version control systems. In this way, the necessary data sources can be constructed in order to apply the congruence framework in non-software development contexts.

REFERENCES

Allen, T.J. (1977). *Managing the Flow of Technology*. MIT Press.

Ancona, D.J. and Calwell, D.F. (1992). Bridging the boundary: external activity and performance in organizational teams. *Administrative Science Quarterly*, Vol. 37.

Arisholm, E., Briand, L.C. and Foyen, A. (2004). Dynamic Coupling Measurement for Object-Oriented Software. IEEE Transactions on Software Engineering, Vol. 30, No. 8, pp. 491-506.

Baldwin, C.Y. and Clark, K.B. (2000). *Design Rules: The Power of Modularity*. MIT Press.

Baldwin, T.T., Bedell, M.D. and Johnson, L.T. (1997). The Social Fabric of a Team-Based MBA Program: Network Effects on Student Satisfaction and Performance. *Academy of Management Journal*, Vol. 40, No. 6, pp. 13690-1397.

Bellotti, V., Ducheneaut, N., Howard, M., Smith, I. (2003). Taking email to task: the design and evaluation of a task management centered email tool. In *Proceedings International Conference on Human Factors in Computing Systems (CHI'03)*, Ft. Lauderdale, FL.

Basili, V.R. and Perricone, B.T. (1984). Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol. 12, No. 1, pp. 42-52.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice, 2nd Edition*. Addison Wesley Publishing.

Bass, M., Bass, L., Herbsleb, J.D. and Cataldo, M (2006). Architectural Misalignment: an Experience Report. To appear in the *Proceedings of the 6th International Conference on Software Architectures (WICSA '07).*

Bass, M., Cataldo, M., Herbsleb, J.D. and Bass, L.J. (2007). *The Impact of Architecture on Coordination: An Empirical Study*. Manuscript, Institute for Software Research, School of Computer Science, Carnegie Mellon University.

Bonacich, P. (1987). Factoring and Weighting Approaches to Status Scores and Clique Identification. *Journal of Mathematical Sociology*, Vol. 2, pp. 113-120.

Booch, G. and Brown, A.W. (2003) Collaborative Development Environments. *Advances in Computers,* Vol. 59, Academic Press.

Borgatti, S.P. (2006). Identifying Sets of Key Players in a Social Network. *Computational and Mathematical Organizational Theory*, Vol. 12, No. 1, pp. 21-34.

Borgatti, S.P. and Everett, M.G. (1999). Models of Core/Periphery Structures. *Social Networks*, 21, pp. 375-395.

Briand, L.C., Wust, J., Daly, J.W. and Porter, D.V. (2000). Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *The Journal of Systems and Software*, Vol. 51, pp. 245-273.

Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition).* Addison Wesley.

Brown, S.L. and Eisenhardt, K.M. (1995) Product Development: Past Research, Present Findings, and Future Directions. *Academy of Management Review*, Vol. 20, No. 2.

Burt, R.S. (1992). *Structural Holes: The Social Structure of Competition*. Harvard University Press.

Burton, R.M. and Obel, B. *Strategic Organizational Diagnosis and Design*. Kluwer Academic Publishers, Norwell, MA, 1998.

Cadiz, J.J., Venolia, G.D., Jancke, G., Gupta, A. (2002). Designing and deploying an information awareness interface. In *Proceedings of the Conference on Computer Supported Cooperative Work* (CSCW'02), New York, NY.

Carley, K.M. (2002). Smart Agents and Organizations of the Future. In *Handbook of New Media*. Edited by Lievrouw, L. and Livingstone, S., Sage, Thousand Oaks, CA.

Carley, K.M., Dombroski, M., Tsvetovat, M., Reminga, J. and Kamneva, N. (2003). Destabilizing Dynamic Covert Networks. In *Proceedings of the 8th International Command and Control Research and Technology Symposium*, National Defense War College, Washington, DC.

Carley, K., Lee, J. and Krackhardt, D. (2001). Destabilizing Networks. *Connections*, Vol. 24, No. 3, pp. 31-34.

Carley, K.M. and Lin, Z. (1995). Organizational Designs Suited for High Performance Under Stress. *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 25, No. 2, pp. 221-230.

Carley, K.M and Ren, Y. Tradeoffs between Performance and Adaptability for C^3I Architectures. *In Proceedings of the 6th International Command and Control Research and Technology Symposium*, Annapolis, Maryland, 2001.

Cataldo, M., Wagstrom, P, Herbsleb, J.D. and Carley, K.M (2006). Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the Conference on Computer Supported Cooperative Work* (CSCW'06), Banff, Alberta, Canada.

Cataldo, M., Bass, M, Herbsleb, J.D. and Bass, L (2007). On Coordination Mechanism in Global Software Development. In *Proceedings of the International Conference on Global Software Engineering*, Munich, Germany.

Cheng, L., Hupfer, S., Ross, S. and Patterson, J. (2003). Jazzing up Eclipse with Collaborative Tools. In *Proceedings of 2003 OOPSLA Workshop on Eclipse Technology Exchange*, New York, New York.

Chidamber, S.R. and Kemerer, C.F. (1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493.

Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Sttaford, J. (2002). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, New York, NY.

Conway, M.E. (1968). How do committees invent? *Datamation*, Vol. 14, No. 5, 28-31.

Crowston, K.C. (1991). *Toward a Coordination Cookbook: Recipes for Multi-Agent Action*. Ph.D. Dissertation, Sloan School of Management, MIT.

Curtis, B. (1981). *Human Factors in Software Development*. Ed. by Curtis, B., IEEE Computer Society.

Curtis, B., Kransner, H. and Iscoe, N. (1988). A field study of software design process for large systems. *Communications of ACM,* Vol. 31, No. 11, pp. 1268-1287.

Daft, R.L. and Weick, K.E. (1984). Towards a model of organizations as interpretation systems. *Academy of Management Review,* Vol. 9, No. 2, pp. 284-295

DeSanctis, G., Staudenmeyer, N. and Wong, S. (1999). Interdependence in Virtual Organizations. In *Trends in Organizational Behavior, Volume 6*, Cooper, C.L. and Rousseau, D.M. Editors, John Wiley & Sons.

de Souza, C.R.B. (2005). *On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support*. Ph.D. dissertation, Donald Bren School of Information and Computer Sciences, University of California, Irvine.

de Souza, C. (2006) Exploring the relationship of software dependencies and the coordination of software development work. In *Proceedings of the Workshop on Supporting the Social Side of Large-Scale Software Development*, Banff, Canada.

de Souza, C.R.B., Quirk, S., Trainer, E. and Redmiles, D. (2007). Supporting Collaborative Software Development thrugh the Visualization of Socio-Technical Dependencies. In *Proceedings of the Conference on Supporting Group Work (GROUP'07)*, Sanibel Island, FL.

de Souza, C.R.B., Redmiles, D., Cheng, L., Millen, D. and Patterson, J. (2004). How a Good Software Practice Thwarts Collaboration – The multiple roles of APIs in Software Development. In *Proceedings of the 12th Conference on Foundations of Software Engineering (FSE '04)*, Newport Beach, CA, 221-230.

Eppinger, S.D., Whitney, D.E., Smith, R.P. and Gebala, D.A. (1994). A Model-Based Method for Organizing Tasks in Product Development. *Research in Engineering Design*, Vol. 6, pp. 1-13.

Espinosa, J.A. (2002). *Shared Mental Models and Coordination in Large-Scale, Distributed Software Development*. Unpublished Ph.D. Dissertation, Graduate School of Industrial Administration, Carnegie Mellon University.

Fenton, N.E. and Neil, M. (1999). A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, Vo. 25, No. 5, pp. 675-689.

Fenton, N.E., Pfleeger, S.L. and Glass, R.L. (1994). Science and Substance: A Challenge to Software Engineers. IEEE Software, Vol. 11, No. 4, pp. 88-95.

Fisher, D., Brush, A.J., Gleave, E. and Smith M.A. (2006). Revisiting Whittaker and Sidner's "Email Overload": Ten Years Later. In *Proceedings of the Conference on Computer Supported Cooperative Work* (CSCW'06), Banff, Alberta, Canada.

Frantz, T., Cataldo, M. and Carley, K.M. (2007). *Measuring Robustness under Uncertainty: Topology Matters Too.* Manuscript, Institute for Software Research, School of Computer Science, Carnegie Mellon University.

Freeman, L.C., Romney, A.K. and Freeman, S.C. (1987). Cognitive Structure and Informant Accuracy. *American Anthropologist*, Vol. 89, pp. 310-335.

Freeman, L.C. (1977). A Set of Measures of Centrality based on Betweeness. *Sociometry*, Vol. 40, pp. 35-41.

Freeman, L.C. (1979). Centrality in Social Networks: I. Conceptual Clarification. *Social Networks*, Vol. 1, No. 3, pp. 215-239.

Galbraith, J.R. (1973) *Designing Complex Organizations*. Addison-Wesley Publishing.

Gall, H. Hajek, K. and Jazayeri, M. (1998). Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, Bethesda, Maryland.

Garcia, A., et al. (2007). Assessment of Contemporary Modularization Techniques, ACOM'07 Workshop Report. *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 5, pp. 31-37.

Gersick, C.J. (1988). Time and transition in work teams: Toward a new model of group development. *Academy of Management Journal,* Vol. 31, pp. 9-41.

Gersick, C.J. (1989). Marking time: Predictable transitions in task groups. *Academy of Management Journal,* Vol. 32, pp. 274-309.

Geyer, W., Brownholtz, B., Muller, M., Dugan, C., Wilcox, E. and Millen, D.R. (2007). Malibu Personal Productivity Assistant. In *Proceedings International Conference on Human Factors in Computing Systems (CHI'07) – Work in Progress Section*, San Jose, CA.

Graves, T.L., Karr, A.F., Marron, J.S. and Siy, H. (2000). Predicting Fault Incidence Using Software Change History, *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 653-661.

Grinter, R.E., Herbsleb, J.D. and Perry, D.E. (1999). The Geography of Coordination Dealing with Distance in R&D Work. In *Proceedings of the Conference on Supporting Group Work (GROUP'99)*, Phoenix, Arizona.

Handley, H.A.H. and Levis, A.H. (2001). A Model to Evaluate the Effect of Organizational Adaptation. *Computational and Mathematical Organizational Theory*, Vol. 7, No. 1, pp. 5-44.

Hassan, A.E. and Holt, R.C. (2004). C-REX: An Evolutionary Code Extractor for C. *CSER Meeting*. Canada, 2004

Hauschildt, J. and Schewe, G. (2000). Gatekeeper and process promoter: key persons in agile and innovative organizations. *International Journal of Agile Management Systems*, Vol. 2, pp. 96-103

Henderson, R.M. and Clarck, K.B. (1990). Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Administrative Science Quarterly*, Vol. 35, pp. 9-30.

Herbsleb, J.D. and Mockus, A. (2003). An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering,* Vol. 29, No. 6, pp.

Herbsleb, J.D.and Moitra, D. (2001).Global Software Development. *IEEE Software*, March/April, pp. 16-20.

Herbsleb, J.D., Mockus, A. and Roberts, J.A. (2006). Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proceedings of the International Conference on Information Systems (ICIS '06)*, Milwaukee, Wisconsin.

Herbsleb, J.D., Mockus, A., Finholt, T.A., and Grinter, R.E. (2000). Distance, Dependencies, and Delay in a Global Collaboration. *In Proceedings of the Conference on Computer Supported Cooperative Work* (CSCW'00), Philadelphia, Pennsylvania.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 1, 26-60.

Hutchens, D.H. and Basili, V.R. (1985). System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, Vol. 11, No. 8, pp. 749-757.

Jazz Project (2007). http://jazz.net/pub/index.jsp. URL accessed on October 25th, 2007.

Jones, C. (1991). *Applied Software Measurement*, McGraw-Hill.

Kan, S.H. (2002). *Metrics and Models in Software Quality Engineering*, Addison-Wesley.

Karolak, D.W. (1998). *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society.

Krackhardt, D. (1988). Predicting with networks: nonparametric multiple regression analysis of dyadic data. *Social Networks*, Vo. 10, pp. 359-381.

Krackhardt, D. and Carley, K.M. (1998). A PCANS Model of Structure in Organization. In *Proceedings of the 1998 International Symposium on Command and Control Research and Technology*, pp.113-119.

Kraut, R.E. and Streeter, L.A. (1995). Coordination in Software Development. *Communications of ACM,* Vol. 38, No. 3, pp. 69-81.

Kwan, I., Damian, D. and Storey, M.A (2006). Visualizing a Requirements-centered Social Network to Maintain Awareness within Development Teams. In *Proceedings of the 1st Workshop on Requirements Engineering Visualization (REV'06)*, Minneapolis, MN.

Leffingwell, D. and Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach, 2nd Edition*. Addison-Wesley.

Levchuk, G.M. et al. (2004). Normative Design of Project-Based Organizations – Part III: Modeling Congruent, Robust and Adaptive Organizations. *IEEE Trans. on Systems, Man & Cybernetics*, Vol. 34, No. 3, pp. 337-350.

Malone, T.W. and Crownston, K. (1994). The interdisciplinary study of coordination. *Comp. Surveys*, Vol. 26, No. 1, pp. 87-119.

March, J.G and Simon, H.A. (1958). *Organizations*. Wiley, New York, NY.

McDonough, E.F., Kahn, K.B. and Barczak, G. (2001). An Investigation of Global, Virtual and Colocated New Product Development Teams. *Journal of Product Innovation Management*, Vol. 18, pp. 110-120.

McGrath, J.E. (1984). *Groups: Interaction and Performance*, Prentice-Hall, Englewood Cliffs, NJ.

Minto, S. and Murphy, G. (2007). Recommending Emergent Teams. In *Proceedings of the 4th Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN.

Mintzberg, H. (1979). *The Structuring of Organizations: A Synthesis of the Research*. Prentice-Hall, Englewood Cliffs, NJ.

Mockus, A. and Weiss, D.M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, Vol. ??, No. ??, pp. 169-180.

Moeller, K.H. and Paulish, D. (1993). An Empirical Investigation of Software Fault Distribution. In *Proceedings of the International Software Metrics Symposium*, IEEE CS Press.

Morelli, M.D., Eppinger, S.D., and Gulati, R.K. (1995). Predicting Technical Communication in Product Development Organizations. *Transactions on Engineering Management*, Vol. 42, No. 3.

Murphy, G.C., Notkin, D., Griswold, W.G. and Lan, E.S. (1998). An empirical study of call graph extractors. *ACM Transactions on Software Engineering Methodology*, Vol. 7, No. 2, pp. 158-191.

Mutton, P. (2004). Inferring and visualizing social networks on Internet Relay Chat. *In Proceedings of the Information Visualization Conference (IV '04)*.

Nagappan, N and Ball, T (2007). Explaining Failures Using Software Dependencies and Churn Metrics. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain.

Olson, G.M. and Olson, J.S. (2000). Distance Matters. *Human-Computer Interaction*, Vol. 15, No. 2 & 3, pp. 139-178,

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of ACM,* Vol. 15, No. 12, 1053-1058.

Paulk, M.C., Weber, C.V., Curtis, B. and Chrissis, M.B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Software Engineering Institute Series in Software Engineering, Addison-Wesley.

Perdu, D.M. and Levis, A.H. (2001). Adaptation as a Morphing Process: A Methodology for the Design and Evaluation of Adaptive Command and Control Teams. *Computational and Mathematical Organizational Theory*, Vol. 4, No. 1, pp. 5-41.

Podgurski, A. and Clarke, L.A. (1990). A Formal Model for Software Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering,* Vol. 16, No. 9, pp. 965-979.

Pressman, R.S. *Software Engineering: A Practitioner's Approach*, McGraw-Hill.

Sangwan, R. et al. (2006). *Global Software Development Handbook*, Auerbach Publishers.

Sarma, A. (2005). *A Survey of Collaborative Tools in Software Development*. ISR Technical Report #UCI-ISR-05-3, Donald Bren School of Information and Computer Sciences, University of California, Irvine.

Sarma, A., Noroozi, Z. and van der Hoek, A. (2003). Palantir: Raising Awareness among Configuration Management Workspaces. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*.

Selby, R.W. and Basili, V.R. (1991). Analyzing Error-Prone System Structure. *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, pp. 141-152.

Schummer, T. and Haake, J.M. (2001). Supporting Distributed Software Development by Modes of Collaboration. In *Proceedings of the European Conference on Computer-Supported Collaborative Work (ECSCW '03*).

Simon, H.A. (1962). The Architecture of Complexity. *In Proceedings of the American Philosophical Society*, Vol. 106, No. 6, pp. 467-482.

Simon, H.A. (1996). *The Sciences of the Artificial*, MIT Press.

Singer, J.D. and Willet, J.B. (2003). *Applied Longitudinal Data Analysis*. Oxford University Press.

Sosa, M.E., Eppinger, S.D., and Rowles, C.M. (2004). The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, Vol. 50, No. 12, pp. 1674-1689

Sparrowe, R.T., Liden, R.c, Wayne, S.J. and Kraimer, M.L. (2001). Social networks and the performance of individuals and groups. Academy of Management Journal, Vol. 44, No. 2, pp. 316-325.

Staudenmayer, N. (1997). *Managing Multiple interdependencies in Large Scale Software Development Projects*. Unpublished Ph.D. Dissertation, Sloan School of Management, Massachusetts Institute of Technology, Stevens, W.P., Myers, G.J. and Constantine, L.L. (1974). Structure Design. IBM Systems Journal, Vol. 13, No. 2.

Sullivan, K.J., Griswold, W.G., Cai, Y, and Hallen, B. (2001). The Structure and Value of Modularity in Software Design. *In Proceedings of the International Conference on Foundations of Software Engineering* (FSE '01), Vienna, Austria, 99-108.

Thompson, J.D. (1967). *Organizations in Action: Social Science Bases of Administrative Theory*. McGraw-Hill, New York, NY.

Trainer, E., Quirk, S., de Souza, C. and Redmiles, D. (2005). Bridging the Gap between Technical and Social Dependencies with Ariadne. In *Proceedings of Workshop on the Eclipse Technology Exchange*, San Diego, California.

Von Hippel, E. (1990). Task Partitioning: An Innovation Process Variable. *Research Policy*, Vol. 19, pp. 407-418.

Wattenberg, M., Rohall, S., Gruen, D. and Kerr, B. (2005). E-Mail Research: Targeting the Enterprise. *Journal of Human-Computer Interaction*, Vol. 20, pp. 139-162.

Yassine, A., Joglekar, N., Braha, D., Eppinger, S. And Whitney, D. (2003). Information Hiding in Product Development: The Design Churn Effect. Research in Engineering Design, Vol. 14, pp. 145-161.

Yu, L. (2006). Understanding Component Co-evolution with a Study on Linux. *Empirical Software Engineering*, Vol. 12, pp. 123-141.

Zimmerman, T., and Weibgerber, P. (2004). Preprocessing CVS Data for Fine-grain Analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, Scotland, U.K.

Zimmerman, T., Weibgerber, P. Diehl, S. Zellers, A. (2005). Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 429-445.

APPENDIX A: SURVEY FOR PROJECT D

The following survey was used in project D to collect information about coordination behavior as well as self-assessed performance data. The survey was administered twice in two consecutive development iterations. The administration of the survey was performed over the internet. The survey resided on a server located at Carnegie Mellon's campus and each respondent logged-in to the survey after being authentificated against project records stored in a database.

Question 1:

```
      In the period between DATE X AND Y, if you have spent time at the
following sites, please indicate the number of working days you have
spent at the following sites:

      COMPANY Office at LOCATION 1
      COMPANY Office at LOCATION 2
      COMPANY Office at LOCATION 3
      COMPANY Office at LOCATION 4
      Other COMPANY Office, please indicate name:
```

Question 2:

```
      In the period between DATE X AND Y, if you have interacted with a
person, please select "yes" next to the team that they belong to, and
then indicate how often you have communicated with this person
regarding integration and development</u></b>-related activities:

      Please select "yes" for any team that contains team members that
you have interacted with during the specified period:


      TEAM 1  :  YES / NO
      ....
      TEAM 14 :  YES / NO
```

Note: a pop-up window with the team's roster would appear if "yes" was selected.

Question 3:

```
      Please select the option that best describes your agreement with
the statement of the questions. If you have not performed any
```

integration or development work, please select the 'Does not apply' option.

A. I am satisfied with the progress I have made on integration-related tasks in the period between DATE X AND Y:

```
Strongly disagree  (value = 1)
Disagree           (value = 2)
Agree              (value = 3)
Strongly agree     (value = 4)
Does not apply     (value = 0)
```

B. I am satisfied with the progress I have made on development-related tasks in the period between DATE X AND Y:

```
Strongly disagree  (value = 1)
Disagree           (value = 2)
Agree              (value = 3)
Strongly agree     (value = 4)
Does not apply     (value = 0)
```

Question 4: Open-ended question

A. In your opinion, what was the biggest challenge in working with people who were in the same location as you in the period between DATE X AND Y:

B. In your opinion, what was the biggest challenge in working with people who were NOT in the same location as you in the period between DATE X AND Y:

C. In your opinion, what action, if any, could be taken to improve the team's effectiveness in developing and delivering software in the period between DATE X AND Y: