UNIVERSITY OF CALIFORNIA,
IRVINE

On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Information and Computer Science

by

Cleidson Ronald Botelho de Souza

Dissertation Committee:
Professor David F. Redmiles, Chair
Professor Paul Dourish
Professor André van der Hoek

2005
The dissertation of Cleidson Ronald Botelho de Souza
is approved and is acceptable in quality
and form for publication on microfilm:

________________________________________

________________________________________

________________________________________

Committee Chair

University of California, Irvine
2005
DEDICATION

To my wife.
TABLE OF CONTENTS

List of Figures ........................................................................................................................................... viii
List of Tables ............................................................................................................................................... ix
Acknowledgements ..................................................................................................................................... x
Curriculum Vitae ......................................................................................................................................... xii
Abstract ..................................................................................................................................................... xvi

Chapter 1 - Introduction ............................................................................................................................. 1
  1.1. STRUCTURE OF THE DISSERTATION ............................................................................................... 3

Chapter 2 - Motivation And Research Approach ....................................................................................... 5
  2.1. INTRODUCTION ............................................................................................................................ 5
  2.2. ON SOFTWARE DEPENDENCIES AND COORDINATION ............................................................ 5
       2.2.1. Software Dependencies: Techniques and Principles ................................................................. 5
  2.3. SOFTWARE DEPENDENCIES AND COORDINATION ..................................................................... 7
  2.4. EMPIRICAL STUDIES ................................................................................................................... 7
  2.5. CURRENT LIMITATIONS ............................................................................................................... 8
       2.5.1. Limitations of Empirical Studies ............................................................................................. 8
       2.5.2. Limitations of Approaches and Tools .................................................................................... 10
  2.6. RESEARCH AGENDA .................................................................................................................... 11
  2.7. RESEARCH APPROACH ............................................................................................................... 13
       2.7.1. Research Questions ............................................................................................................... 14
       2.7.2. The MVP and MCW Field Studies ......................................................................................... 15
       2.7.3. The Tool: Ariadne ............................................................................................................... 15
  2.8. CONTRIBUTIONS ............................................................................................................................ 16

Chapter 3 - Literature Review .................................................................................................................... 18
  3.1. INTRODUCTION ............................................................................................................................ 18
Chapter 4 - Field Studies: Settings and Methodology ..................... 47

4.1. METHODOLOGY ........................................................................... 47
4.1.1. Introduction ............................................................................ 47
4.1.2. Site Selection .......................................................................... 47
4.1.3. Data Collection Methods ......................................................... 48
4.1.4. Data Collection Process ........................................................... 50
4.1.5. Data Analysis Methods ............................................................. 52

4.2. ORGANIZATIONAL SETTINGS ...................................................... 55
4.2.1. The MVP Study ..................................................................... 55
4.2.2. The MCW Study .................................................................... 62

Chapter 5 - Dependency Management as Impact Management .......... 71

5.1. INTRODUCTION ........................................................................... 71
5.2. IMPACT MANAGEMENT ................................................................. 71
5.3. IMPACT NETWORK IDENTIFICATION ........................................ 75
5.3.1. MVP ....................................................................................... 76
5.3.2. MCW ....................................................................................... 78
LIST OF FIGURES

Figure 1 – Dependency Types and associated Coordination Mechanisms .................. 36
Figure 2 - Mapping Design Interfaces and Team Interactions .................................. 44
Figure 3 - Example of Coding .................................................................................. 53
Figure 4 - Tyrant Call-Graph .................................................................................... 119
Figure 5 - Ariadne’s screenshot ................................................................................ 120
Figure 6 - A social call graph .................................................................................... 123
Figure 7 - Ariadne’s Architecture ............................................................................ 125
Figure 8 - Ariadne’s API for Programming Dependency Information .................... 127
Figure 9 - Forms of participation: Centralized ........................................................... 133
Figure 10 - Forms of participation: Densely Networked ............................................ 134
Figure 11 - Forms of participation: Core-Periphery .................................................. 135
Figure 12 - “Hawkprime” in periphery .................................................................... 138
Figure 13 - “Hawkprime” shifts to core .................................................................... 138
Figure 14 - “Umagesh” in core ................................................................................ 139
Figure 15 - “Umagesh” shifts to periphery ................................................................. 140
Figure 16 - Tyrant’s sociogram ................................................................................ 144
Figure 17 - MGR Call-graph .................................................................................... 148
Figure 18 - MGR Sociogram .................................................................................... 150
LIST OF TABLES

Table 1 - Summary of Strategies ........................................................................................................... 75
Table 2 – Tool Support for Impact Management Strategies................................................................. 108
Table 3 – Megamek’s Technical Evolution across Three Releases.................................................... 121
ACKNOWLEDGMENTS

After spending these past few years working on my doctorate, I have many people to thank. Before all, though, I want to thank God for His love and for allowing me to achieve this.

First, I want to thank my advisor, David Redmiles, for his guidance. He has not only been my academic advisor since the beginning of my studies at the university, but also a personal advisor on several occasions. Being able to talk to your advisor about something other than your dissertation or the paper due in a few hours is definitely a blessing in this stressful process. For that, I am thankful.

I also want to thank the other members of my committee, André van der Hoek and Paul Dourish. They have provided excellent guidance with respect to my dissertation work, as well as with several other aspects of my research. In particular, I want to thank Paul for his patience with my infinite questions. I have learned a lot from him.

Other researchers with whom I had a chance to interact along these years also deserve a special thanks, especially Gloria Mark because of her advice in some of the research endeavors that I explored. Others who deserve mentioning include John Penix and Maarten Sierhuis at NASA/Ames, and Li-Te Cheng, David Millen, Michael Muller, and John Patterson in the Collaborative User Experience Group at IBM Research.

I cannot forget the UCI staff for their prompt help. They made my life easier. Special thanks go to Kris Bolcer, Debi Brodbeck, and Kiana Fallah.

Gordon, Heather, Briana, Ethan, and Cameron Russel have been my far-away-from-home family along these years. I thank them for their friendship and I thank God for having them in my life.

Along these years, I also discovered new friends and colleagues. I thank the unofficial Brazilian Student Association members for the barbecues, pizzas, movies, conversations, “Brazilian Friday lunches,” parties, and other diversions. I want to thank them for their friendship. No matter which paths our lives take, please do know that you will always be in my heart.

Márcio Dias, Roberto Silveira, Leila Naslavsky, André Nacul, and Rogério de Paula were also colleagues at school and provided valuable insights in several occasions including practice talks, presentations, papers, and so forth. Their help,
even when they did not know me, is greatly appreciated. I have learned to respect them as professionals, but more important, I have learned to love them as friends. I am sure we will always be in touch.

I also owe thanks to several other students in the School of Information and Computer Science for many invaluable discussions that contributed directly or indirectly to this research. I cannot list all students, so I will limit myself to just a few names: Steve Abrams, Ulrik Christensen, Victor Gonzales, Sameer Patil, Stephen Quirk, Erik Trainer, and Steven Tuttle. Interactions with Steve, Victor, Stephen, and Erik were particularly valuable in the final months of this dissertation. In contrast, Ulrik and Steven were very important in the beginning of my studies.

I want to use this opportunity to acknowledge the financial support provided by the Brazilian taxpayers provided as a scholarship by CAPES. I also had funding from the National Science Foundation and IBM. I am grateful to my department at UFPA for the opportunity to be here.

In general, to all those who helped make this possible.

Last, but certainly not least, I want to thank my lovely wife Patrícia for her love, friendship, support, encouragement, understanding, and patience along all these years. I will never be able to thank her enough for that. I will not even be able to express my gratitude to and love for her. My dear parents also were a constant presence in my life, even far away from me. Talking to them kept reminding me of the beautiful things that I was missing and helped to keep my focus. I want to thank them and all my family in Brazil for their support and encouragement. Knowing that they loved me and that they prayed for me, made me stronger in the difficult times I faced.
CURRICULUM VITAE

Cleidson Ronald Botelho de Souza

EDUCATION

Ph.D., Information and Computer Science  October, 2005
University of California, Irvine

M.S., Information and Computer Science  June, 2003
University of California, Irvine

M.S., Computer Science  September, 1998
State University of Campinas

B.S., Computer Science  January, 1996
Federal University of Pará

HONORS

Ph.D. Dissertation Fellowship  2005
Donald Bren School of Information and Computer Sciences

Best Scientific Initiation Work  1996
Life Sciences, Federal University of Pará.

3rd Prize Best Scientific Initiation Work  1995
Science and Technology, Brazilian Society for Science Progress

RESEARCH INTERESTS

Computer-supported cooperative work (CSCW), human-computer interaction (HCI), empirical software engineering, collaborative software engineering.
RESEARCH EXPERIENCE

Faculty
Departamento de Informática
Federal University of Pará
1998 - present

Research Associate
Department of Information and Computer Science
University of California, Irvine
2001 - present

Research Associate
Collaborative User Experience Research Group
IBM T. J. Watson, Cambridge, MA
Summer 2004

Research Associate
Collaborative User Experience Research Group
IBM T. J. Watson, Cambridge, MA
Summer 2003

Research Associate
Automated Software Engineering Group
NASA Ames Research Center
Summer 2002

PEER REVIEWED PAPERS


NON-PEER REVIEWED PAPERS


WORKSHOPS


As software systems provide more, and more distributed, real-time services to our society, it is possible to witness their growing complexity. One way to manage this complexity is to decompose software systems into smaller parts, called modules. The predictable consequence of dividing a system into modules is that these modules need to be put back together in some coordinated way, so that the software system can provide services. A dependency between software modules is said to exist when one module relies on another to perform its operations or when changes to the latter must be reflected on the former.

Dependencies between software modules affect their development, maintenance, and reuse. More important, they affect the coordination of software development efforts. Although this relationship has been long known by researchers and practitioners, it has been largely unexplored. Most researchers focus on the technical aspects of the dependencies – identification, analysis, and maintenance
– instead of focusing on their implications for understanding the collaborative work of software production. Meanwhile, empirical studies of software dependencies focus on how organizations and teams adopt strategies to manage these dependencies.

To address this issue, I have conducted two field studies to understand how software developers manage the effect of these dependencies in the coordination of their work. Using ethnographic data, I detail how management of dependencies can be understood as impact management – the work performed by software developers to minimize the impact of one’s effort on that of others, and at the same time, the impact of others’ efforts on one’s own. The main aspect underlying impact management is used to inform the design of Ariadne, a tool that aims to facilitate this same activity.

Ariadne is evaluated in two different settings, each examined to determine how software dependencies can be used to facilitate the understanding and enactment of collaborative software development activities. This dissertation concludes by using the observations from my field studies and results from my evaluations to suggest implications for empirical software engineering research, organizational work practices, and the design of collaborative technologies.
Chapter 1 - Introduction

As software systems provide more, and more distributed, real-time services to our society, it is possible to witness their growing complexity. One way to manage this complexity is to decompose systems into smaller parts, the subsystems (Simon 1996). In software systems, this idea is called decomposition, and these smaller parts are called modules (Ghezzi, Jazayeri et al. 2003). The predictable consequence of dividing a system into modules is that these modules need to be put back together in some coordinated way so that the software system can provide services. A dependency between software modules is said to exist when a module relies on another to perform its operations or when changes to the latter must be reflected on the former (Spanoudakis and Zisman 2004).

Modules can be more easily understood, extended, modified, and reused if they are less dependent on other modules (Stevens, Myers et al. 1974). Furthermore, software dependencies affect the coordination of software development efforts (Parnas 1972; Grinter 2003; Sosa, Eppinger et al. 2003; de Souza, Redmiles et al. 2004). Simply put, two different software engineers developing two dependent modules need to coordinate their work to make sure their modules are aligned and can properly interact.

That being said, one can acknowledge the importance of the study of software dependencies to software development. In fact, there have been years of study of the relationship between coordination of software development work and software dependencies. Theoretical predictions proposed by Conway (1968) and Parnas (1972) have been corroborated by different empirical studies (Curtis,
Krasner et al. 1988; Grinter 1998; Herbsleb and Grinter 1999; de Souza, Redmiles et al. 2004; MacCormack, Rusnak et al. 2004). However, little research has been attempted with the specific goal of understanding how software developers handle the effect of software dependencies in the coordination of their work. Organizational (Grinter 2003) and team (Staudenmayer 1997) strategies have been identified, but little is known about how these practices are woven into the everyday work of software developers – in short, how software developers manage dependencies. This understanding is missing, yet it is necessary to help software engineers work together more effectively and design information systems that support their work.

This dissertation aims to fill this gap. It describes an empirical study of software developers in two different organizations, highlighting their strategies to manage software dependencies. As I will illustrate through the presentation of ethnographic data, management of dependencies can be understood as impact management, the work performed by software developers to minimize the impact of one’s effort on others and, at the same time, the impact of others on one’s own effort.

The main aspects underlying impact management are used to inform the design of Ariadne, a tool that aims to facilitate this same activity. This dissertation also presents two evaluations of Ariadne that attempt to validate the underlying impact management features that Ariadne provides. Finally, I discuss the implications for organizational work practices and the design of collaborative technologies that arise from the results of the empirical study and evaluations of Ariadne.
1.1. Structure of the Dissertation

The dissertation is organized in the following manner.

Chapter 2 – Motivation and Research Approach – This chapter briefly reviews the literature, identifying important aspects that have not been covered by previous studies. These aspects suggest an overarching research agenda to the study of software dependencies. Part of this agenda is addressed in this dissertation through specific research questions. This chapter presents these questions, the approach used to answer these questions, and the contributions resulting from having these questions answered.

Chapter 3 – Literature Review – This chapter provides a broader review of software dependencies, including approaches, theoretical observations, and empirical studies. One of the key goals of this study is to point out the lack of use of software dependencies in the understanding and execution of the coordination required in software development efforts.

Chapter 4 – Field Studies: Settings and Methodology – This chapter initially describes the methods that I adopted for data collection and analysis in each one of the settings. It then describes the organizational settings in which I conducted the two field studies.

Chapter 5 – Dependency Management as Impact Management – This chapter presents the results of the field studies that I conducted. It illustrates, through the presentation of ethnographic data, how dependency management is woven into software developers’ daily work as impact management. Finally, it discusses how the software architecture and other organizational factors influence these strategies.
Chapter 6 – Using Dependency Analysis with Ariadne – The results of the field studies presented in the chapter 5 are translated into the software tool Ariadne. This chapter describes Ariadne in detail: its design decisions, architecture, main approach, and features.

Chapter 7 – Navigating Mazes of Dependencies – This chapter describes the methods I used to evaluate Ariadne, as well as the settings in which the evaluations occurred. It also describes the results of my evaluation of how Ariadne can facilitate the understanding and execution of software development activities by researchers and practitioners. These results are discussed in light of the previous results of my field studies.

Chapter 8 – Implications and Future Work – Presented here are implications for the design of collaborative tools and organizational practices that result from this dissertation. The chapter concludes with suggestions for further research.

Chapter 9 – Conclusions - This chapter summarizes the major contributions of this dissertation and presents concluding remarks.
Chapter 2 - Motivation and Research Approach

2.1. Introduction

The relationship between software dependencies and coordination of software development efforts lies between two research areas: software engineering and computer-supported cooperative work (CSCW). Whereas software engineering researchers are concerned with the software being developed and the process of developing it, one of CSCW’s main concerns is with the coordination of cooperative work and how computational tools can support this task (Schmidt and Bannon 1992). As a consequence, it is necessary to review both the software engineering and the CSCW literature to properly understand the relationship of interest. This is the goal of this chapter, to briefly review the literature to identify limitations of the current approaches and studies. By identifying these limitations, it is possible to define a research agenda that suggests where further research in the area is necessary. After that, this chapter describes the research questions that I address in this dissertation, the approach that I adopted to answer these questions, and the specific contributions that arise from answering these questions.

2.2. On Software Dependencies and Coordination

2.2.1. Software Dependencies: Techniques and Principles

Software engineers have long recognized the need to deal with dependencies between software components. In fact, they have created several techniques, tools, and principles to deal with them. For example, dependency analysis
techniques have been developed to deal with the different abstractions used in the construction of software systems: programs (Ferrante, Ottenstein et al. 1987; Podgurski and Clarke 1989), components (Vieira and Richardson 2002), and software architectures (Stafford and Wolf 2001). These approaches are implemented in software development tools and share the ultimate goal of identifying software dependencies to facilitate its understanding, maintenance, and reusability.

Another approach adopted by researchers and practitioners is the creation of mechanisms in programming languages to reduce dependencies between software elements. In this case, the most important principle is information hiding (Parnas 1972), which motivates several concepts in programming languages, including interfaces, data encapsulation, and polymorphism (Larman 2001). Similarly, cohesion and coupling (Stevens, Myers et al. 1974; Constantine and Yourdon 1979) are conceptual measures of the degree of dependency within a module or between modules, respectively. These measures are operationalized in metrics, allowing one to measure the quality of a software system (Fenton and Pfleeger 1997).

Dependency relationships have also been studied under the area of software traceability. In this case, instead of focusing on the source code or the software architecture, the focus is on dependency relationships between the artifacts created during software development (Spanoudakis and Zisman 2004). Dependencies between artifacts express that one relies on another or has a potential for impact: whenever an artifact changes, other artifacts might need to change as well.
2.3. Software Dependencies and Coordination

When Parnas proposed the principle of information hiding, he also suggested that such principle would bring a managerial advantage: reducing dependencies between software modules would also reduce developers’ dependencies on one another (Parnas 1972). Therefore, this principle recognizes the relationship between software dependencies and coordination and has been accepted today in the software engineering community\(^1\) (Herbsleb and Grinter 1999; Ghezzi, Jazayeri et al. 2003).

Conversely, but also supporting this relationship between dependencies and coordination, Conway (1968) postulated that the structure of a software system would reflect the communication needs of the people performing the work. MacCormack and colleagues (MacCormack, Rusnak et al. 2004) exemplify Conway’s argument when they compare a commercial and an open source software development project. According to their results, in the commercial project it was possible to build tighter connections between software components because software developers were collocated. Therefore, a tightly coupled software system was developed in contrast to the similar open source project implemented by distributed developers, which contained less strong dependencies.

2.4. Empirical Studies

McCormack’s is just one of the several empirical studies that corroborated the relationship between software dependencies and coordination (Morelli, Eppinger

---

\(^1\) Other communities have also recognized this relationship between product dependencies and coordination (Sanchez and Mahoney 1996).

Among other things, these studies illustrate that developers employ different social and technical strategies to deal with coordination issues caused by software dependencies. For instance, Staudemayer (1997) describes technical (application programming interfaces, or APIs for short) and social (having a liaison person to handle external dependencies) strategies adopted by six different teams to deal with dependencies. Whereas Staudemayer focuses on team strategies, Grinter (2003) describes mostly organizational strategies adopted by software development companies. An example is the establishment of a building and release team responsible for providing daily builds of the application.

2.5. Current Limitations

With the ultimate goal of understanding the relationship between software dependencies and the coordination of software development work, I analyzed the approaches and empirical studies described in the previous section. This analysis allowed me to identify several limitations of these studies\(^2\). This section describes these limitations.

2.5.1. Limitations of Empirical Studies

Most empirical studies of dependencies do not focus on the individual software developers’ strategies to handle dependencies, that is, the ways in which software developers deal with the effects of software dependencies in the

\(^2\) A different perspective, of course, would identify other aspects.
coordination of their work. Although it is true that some of the organizational and team strategies described by other authors are ultimately performed by individuals (e.g., the liaison developer suggested by Staudenmayer (1997)), the analytical focus of these authors is not at the individual level. By not focusing on developers’ practices, it is more difficult to understand how these practices are embedded into software developers’ daily work, a requirement to understand the weaknesses and strengths of the tools used by software developers.

Even when technical approaches used by individual developers are investigated (such as APIs in (Staudenmayer 1997; Herbsleb and Grinter 1999)), the point of view of individual software developers is not explored because authors assume that coordination is simpler because of the usage of interfaces. In fact, work in open implementation (Kiczales 1996; Kiczales, Lamping et al. 1997) suggests that a software component’s interface might not provide enough details about the component’s implementation; other details still need to be communicated, coordinated, and negotiated among the interested parties. This suggests that, despite the usage of interfaces, software developers still need to collaborate to be able to get their work accomplished. In short, the coordination facility provided by interfaces is to be studied, not assumed.

In order to fully inform the design of more adequate tools that facilitate collaborative software development efforts, this understanding of individual

---

3 An exception to this is Grinter’s analysis of configuration management tools to support coordination (Grinter 1995). Grinter studied developers’ practices around configuration management tools only. What I describe in this dissertation is a study of the individual practices adopted by software developers while they employ the several tools that they use to perform their work.
strategies is necessary. This translates into the need to conduct further research aiming to understand how software developers manage the effects of software dependencies in their work. In other words, it is necessary to find out how software developers manage dependencies, how this management is woven into their daily work, and how different tools support (or hinder) this activity.

2.5.2. Limitations of Approaches and Tools

Despite the empirical evidence of the relationship between software dependencies and coordination, most software engineering tools that focus on software dependencies do not leverage the potential for facilitating coordination. Their main goal is to analyze software systems to identify its dependencies and consequently facilitate its understanding, maintenance, and reusability. For instance, tools like jDepend⁴ are used to calculate coupling metrics that inform the quality of a software design (Fenton and Pfleeger 1997). Other, more recent tools support the visualization of metrics to support the evolution of software systems (Collberg, Kobourov et al. 2003; Pinzger, Gall et al. 2005).

Similarly, groupware tools support different social aspects (e.g., negotiation, communication, and coordination); however, most of them do not make use of dependency information. In not doing so, these tools (and the underlying approaches) do not fully leverage their potential for facilitating cooperative efforts.

⁴ http://www.clarkware.com/software/JDepend.html
2.6. Research Agenda

The discussion in the previous section identified several limitations in the approaches and empirical studies focusing on software dependencies. These limitations can be understood as gaps in our understanding of collaborative software production and can be used to define a research agenda suggesting open research issues in the area. The goal of this section is to present this agenda.

*Individual strategies used by software developers to handle software dependencies* – Based on what I presented in the previous section, I argue that we need to discover the set of practices or strategies that individual software engineers employ to manage technical dependencies – not only which strategies, but also when, why, and how these strategies are employed, extended, and/or avoided. One example of such an approach is the usage of interface specifications (Staudenmayer 1997; Grinter, Herbsleb et al. 1999) because they minimize dependencies between software components, facilitating the coordination of group work. Despite that, it is still important to investigate the communication, decision making, and negotiation that software developers engage in because interface specifications are limited.

*The contextual factors that influence these strategies* – We need to conduct studies of software developers’ practices at different organizations at different stages of product development for several domains, that is, on different organizations during varied project phases that develop different products. By looking at different software applications in development, engineers, teams, and organizations, one can potentially contrast contextual factors (e.g., organizational environment, software architecture, and so on) to find out how these factors
influence and are influenced by software developers’ strategies. In short, another important aspect that needs to be sought is an understanding of which and how contextual factors influence the strategies used for dependency management. This can be done only by contrasting results from studies conducted in different situations.

The strengths and weaknesses of collaborative and software development tools – Once the strategies adopted by software engineers are identified, it is possible to identify the strengths and weaknesses of the current tools in supporting these strategies and their underlying goals. In this case, collaborative tools (e.g., instant messenger systems, email clients, etc.) as well as software development tools (e.g., configuration management and bug-tracking tools) are included. By understanding this aspect, we can more adequately (re)design tools that better support software developers’ strategies. Furthermore, understanding the contextual factors that influence these strategies can potentially be leveraged to allow a customization of these tools to match their contexts of use. Thereby, these tools can be made potentially more useful for software developers.

On the design of collaborative software development tools – Software development tools that use dependency information do not leverage the potential to use these techniques to understand and facilitate the coordination of software development activities. Similarly, most groupware tools do not use dependency information to facilitate the coordination of cooperative work. Therefore, I argue that researchers in collaborative software development need to explore the construction of collaborative and/or software development tools that leverage dependency information among artifacts to facilitate the collective effort. With these tools, it is then possible to ask what insights we can gain about
collaborative software development activities by analyzing software dependencies.

The co-evolution of software dependencies and coordination – Another area for research that needs to be explored is the study of the co-evolution of software systems with their associated dependencies and the coordination required to maintain these systems. As discussed in section 2.5, some authors focus on studying the evolution of software systems by analyzing measures of coupling in successive releases. The acknowledged relationship between software dependencies and coordination suggests that one could also study the co-evolution of these two aspects. That is, in addition to studying these aspects independently, we should focus on studying these aspects as they co-evolve because changes in one potentially cause changes in the other. For instance, refactoring techniques can be used to make changes in the code to make it easier to maintain. By changing the overall dependencies among components in the software, refactoring techniques potentially change communication and coordination needs among software developers, impacting positively or negatively the collaborative work.

2.7. Research Approach

The broader research focus of this work is on understanding the relationship between software dependencies and coordination. Such a focus cannot be explored in the limits of a single dissertation; therefore, this dissertation focuses on specific aspects of this larger research. This was accomplished by carefully defining specific research questions to be tackled that would provide important
contributions and by choosing an appropriate research approach to answer these questions. These aspects are described in this section.

2.7.1. Research Questions

To be more specific, the important research questions that my work addresses are:

- What are the individual strategies used by software developers to manage the effect of software dependencies in the coordination of their work? This question aims to address the first topic in the research agenda, so that by looking at practices in different companies, it is possible to understand how these practices are supported – or fail to be supported – by software tools. To answer this question, I conducted two field studies of software development work.

- What value can be derived from using dependency analysis techniques to help software developers to understand and perform their work? This question is to be answered through a proof-of-concept system that automatically identifies technical dependencies among software development artifacts. Again, this is part of the broader research agenda described in the previous section, which aims to augment the design of collaborative software development tools by exploring the usage of dependency information.

In short, to answer these research questions, I adopted two different approaches: fieldwork and design. Each one of these aspects is described below.

---

5 I am using the term “dependency analysis” to represent a range of approaches that allow the identification of dependencies among software artifacts, such as program dependency analysis, software traceability, and so on.
2.7.2. The MVP and MCW Field Studies

To identify and understand the strategies used by software developers to handle software dependencies, I conducted two qualitative studies at different large software development organizations. The first field study was conducted during the summer of 2002, when I was an intern in the MVP team, and the second one was performed during the summer of 2003, when I studied the MCW team. I used observation (Jorgensen 1989) and semi-structured interviews (McCracken 1988) for data collection. Data analysis was conducted by using grounded theory techniques (Strauss and Corbin 1998). My results suggest that dependency management can be seen as impact management; that is, the work required to reduce the impact of other developers’ actions into my own work and the work to reduce the impact of my work into other developers’ work. Impact management is achieved through different technical and social strategies, which are embedded into software developers’ daily work. These results were used to inform the design of a tool, Ariadne, described in the next section.

2.7.3. The Tool: Ariadne

Ariadne is a tool that aims to facilitate the management of dependencies by software developers. It reflects and embodies the results of the field studies that I conducted. For instance, Ariadne allows one to analyze the results of other changes into the context of one’s own work.

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, Ariadne is integrated into this environment and makes use of several of the

---

http://www.eclipse.org
services it provides. In fact, Ariadne uses Eclipse’s SearchEngine class to extract dependencies from a Java project’s source code. It then connects to the configuration management repository associated with this project to retrieve its authorship information. After that, Ariadne uses this information to annotate the dependency graph created by SearchEngine. Finally, it creates and displays a sociogram describing dependency information among software developers, using the graphical framework JUNG (Java Universal Network/Graph Framework).7

To understand how Ariadne can be used to facilitate the coordination of software development efforts, I performed two evaluations of Ariadne. In the first one, I used this tool to explore open source projects to understand the coordination of open source software development work through the usage of dependency analysis. In the second one, I conducted open-ended interviews with both open source and commercial software developers to understand how the results provided by Ariadne could be used to facilitate their work.

2.8. Contributions

By answering the research questions that I proposed, this dissertation presents the following contributions. First, it provides an empirically grounded understanding of software engineers’ individual and team strategies to coordinate their work in face of software dependencies.8 Another contribution of this dissertation is the analytical framework of impact management that I use to

7 http://jung.sourceforge.net
8 I have observed team strategies other than those documented by Staudenmayer (1997) and, when I observe similar strategies, they seem to be deployed in ways that suggest another conceptualization.
analyze and understand these strategies. A third contribution of this dissertation is a discussion about the contextual factors that influence these strategies. Among these factors, the software architecture (as expected) and the organizational environment are the more important ones. In other words, this dissertation illustrates how software developers employ their knowledge about the software dependencies in the architecture to adapt their work. The fourth contribution is the usage of dependency analysis techniques to facilitate the execution and understanding of collaborative software development, which is demonstrated through Ariadne. In order to do that, this dissertation presents an approach for identifying dependencies between developers from dependencies in the code assigned to them, approaches for the organization of open source projects, and finally, the initial evaluation of my approach among commercial and open source software developers.

Before describing these contributions in details, in the following chapter I review the literature, focusing on software dependencies and their relationships to coordination.
Chapter 3 - Literature Review

3.1. Introduction

This chapter starts with the definition of software dependencies that is adopted in this dissertation. This is then followed by a literature review of software dependencies, including approaches, theoretical observations, and empirical studies. Finally, the chapter discusses other related work on coordination and dependencies.

3.2. On the Definition of Dependencies

I surveyed the discipline of software engineering to understand how researchers and practitioners define and use dependency relationships. Based on this survey, I noticed that authors adopt different definitions of dependencies. For instance, Grinter (2003) broadly defines dependencies as “technical relationships in the code,” whereas Wilde (1990) is more specific, defining a dependency as “a relationship between two components so that changes to one may have an impact that will require changes to the other.” Nevertheless, most authors define dependencies in very specific contexts: program dependencies are “syntactic relationships between the statements of a program that represent aspects of the program’s control and data flow” (Ferrante, Ottenstein et al. 1987; Podgurski and Clarke 1989); component dependencies reflect the potential for one component to affect or be affected by the elements that compose the system, including other components (Vieira 2003); and finally, architecture dependencies focus on “both the structural and behavioral relationships among components” that is expressed in architecture description languages (Stafford, Wolf et al. 1998).
Despite the different definitions, dependencies between modules are a consequence of the need for dividing a software system into smaller pieces, an approach called decomposition (Ghezzi, Jazayeri et al. 2003), as well as the need also to combine these modules together, which is necessary to manage the complexity of the (software) system (Simon 1996).

For the purposes of this dissertation, I will adopt the definition of dependency of Spanoudakis and Zisman (2004): a relationship between two software entities e1 and e2, where the existence of e1 relies on the existence of e2, or changes in e2 have to be reflected in e1. I chose this definition because it is broad enough to encompass the other definitions that I surveyed. Furthermore, it includes both dependencies that exist between the same types of artifacts (e.g., two software modules) and dependencies between different types of artifacts (e.g., a requirement and the design that fulfills that requirement). Both aspects need to be studied because they influence software developers’ work. In traceability (Lindvall and Sandahl 1996), where the goal is to model different types of relationships among software artifacts (including dependency), these are called vertical and horizontal relationships, respectively.

A dependency is not a property of the entities themselves, but rather a relationship between them. As a relationship, it can have its own attributes that are independent from both entities’ attributes. The most important of these is the strength or degree of the dependency, which indicates to which extent one entity depends on the other. As I will discuss later, this attribute is important in

---

9 Staudenmayer (1997) calls these component and translation dependencies. She focuses on the effects of these in the coordination of the tasks, which is discussed in section 3.6.
understanding software developers’ work toward software artifacts. That is, a software developer will work differently according to the strength of dependency between the software artifacts.

Dependencies are one-way relations: A depends on B, but B does not depend on A. On the other hand, interdependencies are bi-directional relations: A depends on B, and B depends on A.

3.3. The Effects of Dependencies in Software Development

Software dependencies have been mainly studied in the context of software development due to their implications for the construction and maintenance of software systems (Podgurski and Clarke 1989). For instance, according to Steven et al. (1974):

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous “ripple” effects where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc.

Because of the same reasons, software dependencies reduce the reusability of software modules. A module that is highly dependent on other modules implies that if one wants to reuse this module, he or she will also have to use all the other modules coupled to it.

Software dependencies also have a positive connotation: The existence of a dependency relationship between the requirements and the analysis – and later to a design document that implements this requirement – is something to look
for because it indicates that the software implemented addresses this particular requirement. Some authors even argue that dependencies between requirements can support software reuse: if similar requirements are identified when the stated requirements are compared with existing requirements, then this indicates a possible reusable component (Dahlstedt and Persson 2003).

In the following section, I will describe the main approaches to the study of software dependencies in software engineering.

3.4. Software Dependencies in Software Engineering

3.4.1. Dependency Analysis

Software engineers have long recognized the need to deal with dependencies between software components. For example, dependency analysis techniques have focused on programs (Podgurski and Clarke 1989), component-based systems (Vieira and Richardson 2002), and software architectures (Stafford and Wolf 2001). These approaches are implemented as software development tools with the ultimate goal of helping developers to minimize dependencies to facilitate software reuse, software understanding, and testing activities. For instance, program dependencies are used to improve software testing, fault location when debugging through program slicing (Weiser 1984), maintenance, parallelization, computer security, and code optimization (Podgurski and Clarke 1989). Component dependency analysis is crucial to effective maintenance, evolution, testing, debugging, and management of component-based systems.

---

10 A common approach is the identification of the set of test cases that no longer are applicable due to the changes in the software source-code (Rothermel and Harrold 1997).
(Vieira 2003). In addition, architectural dependency analysis techniques can be used to support architectural reuse, regression testing, and software understanding (Zhao 1997).

Overall, these approaches share the analytical goal of identifying and analyzing dependencies in the software application to be able to reason about its reusability, maintenance, and understanding. They also have in common the usage of the program dependence graphs (also called PDGs) or variations as representations of software dependencies (Ferrante, Ottenstein et al. 1987). According to Horwitz and Reps (1992), formally, a PDG for a program P is a directed graph whose vertices are connected by several kinds of edges. The vertices represent the statements of P, and the edges represent control and data dependencies, which are explained below:

The intuitive meaning of a control dependence edge from vertex v to vertex w is the following: if the program component represented by vertex v is evaluated during program execution and its value matches the label on the edge, then, assuming that the program terminates normally, the component represented by w will eventually execute; however, if the value does not match the label on the edge, then the component represented by w may never execute. (...) Data dependence edges include both flow dependence edges and def-order dependence edges. Flow dependence edges represent possible flow of values, i.e., there is a flow dependence edge from vertex v to vertex w if vertex v represents a program component that assigns a value to some variable x, vertex w represents a component that uses the value of variable x, and there is an x-definition clear path from v to w in the program’s control-flow graph.

For simplicity purposes, software engineering and programming language researchers initially explored the construction of PDGs for programs containing a single procedure or isolated procedures. Later, interprocedural approaches were
explored considering several procedure calls and their parameters and return types (Aho, Sethi et al. 1986). In this case, some authors adopt the term system dependence graph (SDG) instead of PDG. A SDG is made up of a collection of procedure dependence graphs, which are essentially the same as the program dependence graphs defined above, except that they may include interprocedural control and flow dependence edges to represent procedure calls (Horwitz and Reps 1992). System dependence graphs can be used to construct call graphs (Lakhotia 1993) that are used for interprocedural program optimization and program understanding (Murphy, Notkin et al. 1995). According to Callahan and colleagues (1990), a call graph “summarizes the dynamic invocation relationships between procedures. The nodes of the call graph are the procedures in the program. An edge (pl, p2) exists if procedure pl can call procedure p2 from some call site within pl. Hence, each edge may be thought of as representing some call site in the program.”

Despite their different applications in software engineering, Murphy and colleagues discovered that call-graph extractors used in software development tools might provide different results (Murphy, Notkin et al. 1998). The reason why call-graph extractors provide different results compared to those used in compilers is that “software engineering tools place a different – and in some ways more relaxed – set of requirements on call graphs, since the call graphs are most often consumed by humans for the purpose of understanding” (ibid.).
3.4.2. Measures of the Degree of Dependency: Coupling and Cohesion

Cohesion and coupling (Stevens, Myers et al. 1974; Constantine and Yourdon 1979) are concepts used to indicate the degree of dependency within a module or between modules. To be more specific, cohesion is a property of a software module, or program, that represents the functional relatedness of their internal elements (e.g., statements, procedures, and declarations) (Stevens, Myers et al. 1974). It is a measure of the relationships between its elements. The internal cohesion of a module is measured in terms of the strength of binding elements within the module. In other words, cohesion measures the degree of dependencies that occur within a module.

Meanwhile, coupling measures the strength of the dependency relationships between different modules. The term coupling is used as a measure of dependencies between two modules (Stevens, Myers et al. 1974; Ghezzi, Jazayeri et al. 2003). If two modules depend on each other heavily, they are said to have high coupling; otherwise, the modules are said to have low coupling and are almost independent of each other. Coupling between modules occurs under different situations – for instance, if one model refers to the inside of the other, or if modules make use of shared variables or interchange control information (Constantine and Yourdon 1979). These situations create a scale that indicates the strength of the dependency, which is used to classify contexts where coupling is more or less desirable. As I will discuss later, the strength of dependency influences the need for coordination in software (and product) development.
The concepts of coupling and cohesion are interrelated: cohesion refers to the degree of interconnection (or relatedness) among the parts of a single module, and coupling refers to the strength of the interconnections between different modules. Both concepts help establishing the quality of a particular design (Sommerville 2000). In fact, by designing modules with high cohesion and low coupling, maintainability is achieved. That is, if it becomes necessary to change the software, the part to be changed is easily identified because it can be found in a single place. Furthermore, this change does not cause ripple effects. Ideally, modules with high cohesion and low coupling are easier to analyze, understand, modify, test, and reuse (Ghezzi, Jazayeri et al. 2003).

3.4.3. Parnas’s Information Hiding

Another approach adopted by researchers and practitioners is the creation of mechanisms in programming languages to reduce dependencies between software elements. Examples of these mechanisms are data encapsulation, interfaces, and polymorphism used in object-oriented programming languages (Larman 2001). All these mechanisms, however, are motivated by the same principle, Parnas’s information hiding (Parnas 1972). According to this principle, software modules should expose to other modules only the details that are not likely to change, thereby reducing the degree of dependency between these modules. If a particular module’s details are encapsulated within the module and do not affect other modules, they can be changed more easily without causing significant impact. Basically, Parnas proposed a distinction between a module’s interface and implementation. The interface is the stable part, whereas the
implementation is the unstable one. Modules get connected to each other through their interfaces.

As I will discuss later, Parnas also suggested that the information hiding principle brings managerial advantages, facilitating the coordination of software development projects.

3.4.4. Software Traceability

Dependence relationships in software engineering have also been studied under the name traceability. In this case, instead of focusing on the source code or the software architecture, the focus is on dependency relationships between software development artifacts. Software traceability is defined as “the ability to relate artifacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artifacts, and the rationale that explains the form of the artifacts” (Spanoudakis and Zisman 2004).

In a survey of the area, Spanoudakis and Zisman (2004) identified seven possible types of relationships between software artifacts. Dependency is one of them. Moreover, not all authors use the term dependency to refer to this type of relationship; other names used include causal conformance, developmental relations, and correspondence, to name a few.

Traceability was first conceived to describe and follow the life of a requirement. Originally called requirements traceability, “with its emphasis on supporting the customer in ensuring that the requirements agreed upon are met” (Lindvall and Sandahl 1996). Nowadays, however, traceability dependencies also
involve artifacts of different types\textsuperscript{11} potentially created during the different phases of the software development process.

Software traceability and program dependency analysis are the two main approaches that constitute the area of change impact analysis (Bohner and Arnold 1996), an area concerned with “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change” (ibid; p. 3).

3.4.5. System Building in Configuration Management Tools

Configuration management (CM) is the process that controls the changes made to a system and manages the different versions of the evolving software product (Sommerville 2000). CM tools provide several types of services, including versioning, system building, and process support to concurrent engineering, among others. In this section, I am interested in system building because this service is the one that uses information about the dependencies between the products available in the CM system to (re)build the system.

System building is “the process of combining the components of a system into a program which executes on a particular target configuration. This may involve compilation of some components and a linking process, which puts the object code together to make an executable system” (Sommerville 2000). This service facilitated the acceptance of CM systems among software developers because it was useful for them, in contrast to other services that are more useful to the configuration manager (Estublier 2001).

\textsuperscript{11}Note that requirement, program, component, and architecture dependencies involve relationships among artifacts of the same type.
Basically, to (re)build a system, it is necessary to derive objects from other objects (source code). This is done by taking into account building dependencies – rules that define how an object is derived from other objects. Building tools, such as make and ANT, automate the process of system building by using these rules to possibly minimize the number of required recompilations after a change is made. This is possible because such tools analyze the modification date of the source code: if it is later than the modification date of the object code with the same name, then recompilation is necessary. This is the simplest mechanism used by CM tools; other more sophisticated mechanisms can be used as well (Estublier, Leblang et al. 2005). This mechanism potentially reduces the rebuilding time from days to hours, or from hours to minutes. Make and ANT store the building rules in text files, which are generated by CM systems based on the information about the dependencies they have available (Estublier 2001).

3.5. Software Dependencies and Coordination

3.5.1. Theoretical Predictions

Parnas was one of the first researchers to recognize the relationship between software dependencies and coordination. More than 30 years ago, he suggested that by reducing dependencies at the artifact level, it is possible to reduce developers’ dependencies on one another, creating a managerial advantage (Parnas 1972; Herbsleb and Grinter 1999). Currently, this is an accepted argument among researchers and practitioners. In their textbook, Ghezzi et al. describe this as follows:

If a design is composed of highly independent modules, it supports the requirements of large programs: independent modules form the basis of
work assignments to individual team members. The more independent
the modules are the more independently the team members can proceed
in their work. (Ghezzi, Jazayeri et al. 2003, p. 241)
Parnas’s information hiding principle has been applied in other domains in
addition to software engineering. In these fields, it has been called simply
modularity. Modular product design has been adopted by several industries,
including aircraft, automobiles, consumer electronics, and personal computers,
among others (Sanchez and Mahoney 1996). Baldwin and Clark (1997) argue that
the modularity adopted by the computer industry is the key factor for its success.

Organizational science also benefits from modularity because “the creation
of modular product architectures not only creates flexible product design, but
also enables the design of loosely coupled, flexible, ‘modular’ organization
structures” (Sanchez and Mahoney 1996). This is possible only because well-
defined interfaces between the products being developed facilitate coordination
practices, reducing the need for management and control over the module’s
associated personnel (Mintzberg 1979). Some authors even assert that modularity
in the design of products leads to – or at least ought to lead to – modularity in the
design of organizations that produce such products (Langlois 1999).

Conversely, but also supporting this relationship between dependencies
and coordination, Conway (1968) postulated that the structure of a software
system would reflect the communication needs of the people performing the
work. MacCormack and colleagues (MacCormack, Rusnak et al. 2004) exemplify
Conway’s argument when they compare commercial and open source software
development. Because software developers were collocated in the commercial
project, it was easier to build tight connections between software components,
therefore producing a software system more coupled compared to the similar open source project with distributed developers.

In short, while Parnas argued that dependencies shape the coordination and communication activities performed by software developers, Conway argued the opposite, that dependencies reflect these coordination and communication activities. That is, technical dependencies between software components being developed by different developers create a need for communication and coordination between developers, and similarly, the coordination needs of the development tasks are reflected in the software dependencies that are created in the software product.

3.5.2. Empirical Studies

Both Parnas’s and Conway’s arguments have been validated by several different empirical studies. In 1988, for example, Curtis et al. (Curtis, Krasner et al. 1988), in their seminal paper, discussed the impact of fluctuating requirements in the coordination of software development work. Curtis also discussed how the system architecture affected the communication required among project personnel and, at the same time, recognized that “occasionally, the partitioning [of components to reduce dependencies between components] was based not only on the logical connectivity among components, but also on the social connectivity among the staff” (Curtis, Krasner et al. 1988, p. 1280).

More recently, Herbsleb and Grinter (Herbsleb and Grinter 1999) discuss the influence of the software architecture in the coordination of distributed software development. They argue that “the more cleanly separated the modules, the more likely the organization can successfully develop them at different sites”
because this will remove the communication required among the different sites. They conclude:

Attend to Conway’s Law: Have a good, modular design and use it as the basis for assigning work to different sites. The more cleanly separated the modules, the more likely the organization can successfully develop them at different sites.

Finally, Sosa and colleagues (Sosa, Eppinger et al. 2002) uncovered a strong correlation between dependent software entities in a software system and the frequency of communication among the team members dealing with these components. Similarly, ethnographic studies (Staudenmayer 1997; Grinter 2003; de Souza, Redmiles et al. 2004) suggest that technical dependencies among pieces of code create “social dependencies” among software developers. That is, given two dependent pieces of code, the developers responsible for developing those pieces need to interact and coordinate in order to guarantee the smooth flow of work.

These empirical studies also illustrate the different practices used by software developers to deal with software dependencies and their effects on coordination, and suggest that this process is both a technical and a social one. For instance, Staudenmayer (1997) describes technical (such as APIs) and social (having a liaison person to handle external dependencies) strategies adopted by six different teams to deal with dependencies. Other empirical studies (e.g., Grinter, Herbsleb et al. 1999) have also pointed out the importance of APIs in the coordination of distributed software development work. However, they take for granted the independent work allowed by interfaces. de Souza and colleagues (2004) instead unpack exactly how software developers in their daily work activities achieve this independence. Their results are corroborated by work in
open implementation (Kiczales 1996; Kiczales, Lamping et al. 1997), which suggests that interfaces can reduce the coordination burden to only some extent. A software component’s interface might not provide enough details about the component’s implementation, and these details might be necessary to the component user to make appropriate decisions about whether to use the component and how to use it. Other details of a component still need to be communicated, coordinated, and negotiated among the interested parties. As I discussed in the previous chapter, only through an investigation of software developers’ individual work can one understand the strengths and limitations of interfaces in the coordination of software development work.

3.6. Dependencies between Tasks

Researchers in other disciplines, such as CSCW and organizational and management sciences, have also studied the relationship between dependencies in artifacts and coordination. However, instead of focusing directly on the dependencies between the artifacts, these authors focus on the dependencies between the tasks that are required to manipulate these artifacts. By doing so, they argue that they simplify their approaches and still have a powerful framework for the study of coordination. This is best exemplified by the work of Malone and Crowston (1994), which is part of a long tradition in organizational science initiated by Thompson (1967) and extended by several other authors (Van de Ven, Delbecq et al. 1976; Adler 1995). This related work is discussed below.
3.6.1. Thompson’s Organizational Dependencies

In organizational science, task dependencies are often associated with the coordination mechanisms required to deal with them. The degree of the dependency between the tasks is taken into account, so that the more dependent two tasks are, the more difficult is to coordinate them. Thompson’s (1967) seminal work was the first to propose this relationship between the types of dependency and the coordination mechanisms used to deal with them. He defined three types of dependencies (pooled, sequential, and reciprocal), as well as their associated coordination mechanisms (standardization or rules, plans and schedules, and mutual adjustment, respectively).

Thompson also argued that organizations structure themselves in such a way to minimize the coordination costs required. Therefore, the lowest-level units of an organization will group to deal with the more dependent tasks. Then, higher-groups are formed to handle other dependent tasks in the organization. Finally, groups, if necessary, are formed to handle any remaining loosely coupled tasks. Thereby, organizations create a hierarchy of units, similar to a process of successive clustering. Mintzberg (1979) called this criterion for unit grouping in organizations work-flow interdependencies. This happens when groups of operating tasks are organized to reflect the sequence of tasks that need to be performed. Mintzberg also discussed other criteria in his work.

More recently, Adler (1995) has studied how the coordination mechanisms evolved alongside the project, using a temporal perspective. Adler’s work is one of the few that takes into account temporal aspects while analyzing work in task
dependencies. Note also that the temporal aspect of interdependencies arose from his data. As he describes it:

as the phases of work unfold within a time-bound project, departments typically experience different degrees and types of dependencies, and they interact with varying intensities and via different coordination mechanisms. And as a result, in the course of a product development project, neither interdepartmental interdependencies nor coordination mechanisms are constant over time.

Drawing on data from his fieldwork in different organizations, Adler developed a taxonomy of coordination mechanisms and project phases that describes which coordination mechanisms are used in each phase of a project’s process. This taxonomy was later translated into a normative theory.

Staudenmayer (1997) has also extended the work of Thompson. She conducted interviews at two large software development corporations to classify the types of dependencies (between tasks) that software developers face in their daily work. One of these types of dependencies arises from the dependencies in the software architecture. Other types of dependencies include dependencies because of shared artifacts, because particular tools are required for a particular task, because of relationships between the software and the documentation associated with it, and so on. Staudenmayer also provides a quantitative analysis of the frequency each type of dependency was reported. Dependencies between tasks that exist because of dependencies between software components are the ones most reported by software developers. Those are dependencies between the same type of artifacts, code. The frequency of the dependencies between different types of software artifacts is spread among the other types of dependencies that Staudenmayer uses, and consequently cannot be correctly identified.
3.6.2. Coordination Theory

Malone and Crownston (1994) summarized the body of knowledge about coordination in the early nineties under the name coordination theory. Coordination theory draws results from different fields, such as computer science, organization theory, economics, biology, sociology, social psychology, linguistics, law, and political science.

Coordination is defined as the management of dependencies between tasks or activities, therefore acknowledging that the long history of emphasizing dependencies started with Thompson. Coordination theory is based on study of the types of dependencies that might exist between two tasks, and consequently the types of coordination mechanisms used to manage them. In fact, Malone and Crownston argue that it is not necessary to study dependencies between components or artifacts because they “explicitly or implicitly, affect the performance of some activities”. Therefore, viewing dependencies as solely occurring between activities simplifies the approach.

Dependencies and associated coordination processes in the different disciplines surveyed are classified as managing shared resources, managing producer/consumer relationships, managing simultaneity constraints, and managing task/subtask dependencies. In addition, two other activities are considered: group decision-making and communication. All these approaches are described in Figure 1.
### Figure 1 – Dependency Types and Associated Coordination Mechanisms

Based on this analysis of dependencies, Malone and Crowston apply this coordination perspective in different situations – for instance, in the design of cooperative support tools and to facilitate the understanding of the effects of information technology in organizations. Finally, they suggest several aspects where coordination theory needs to be expanded.

#### 3.6.3. Dependencies between Tasks in Software Engineering

Software engineering researchers have also looked at the relationship between task dependencies and coordination. This has been done mostly in the area of software processes. Software development processes are defined as “the coherent set of policies, organizations, structures, technologies, procedures, and artifacts that are needed to conceive, develop, and maintain a software product” (Fuggetta 2000). These processes describe the steps that need to be performed during a software development effort, given the problem to be solved, the

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Examples of coordination processes for managing dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared resources</td>
<td>&quot;First come/first serve&quot;, priority order, budgets, managerial decision, market-like bidding</td>
</tr>
<tr>
<td>Task assignments</td>
<td>(same as for &quot;Shared resources&quot;)</td>
</tr>
<tr>
<td>Producer / consumer relationships</td>
<td></td>
</tr>
<tr>
<td>Prerequisite constraints</td>
<td>Notification, sequencing, tracking</td>
</tr>
<tr>
<td>Transfer</td>
<td>Inventory management (e.g., &quot;Just In Time&quot;, &quot;Economic Order Quantity&quot;)</td>
</tr>
<tr>
<td>Usability</td>
<td>Standardization, ask users, participatory design</td>
</tr>
<tr>
<td>Design for manufacturability</td>
<td>Concurrent engineering</td>
</tr>
<tr>
<td>Simultaneity constraints</td>
<td>Scheduling, synchronization</td>
</tr>
<tr>
<td>Task / subtask</td>
<td>Goal selection, task decomposition</td>
</tr>
</tbody>
</table>

36
specific development project, and all particularities of the organization, environment, and software product being developed.

Software processes can be used for many purposes, such as facilitating human communication and understanding, learning and training newcomers, and automating process guidance and execution. However, in order to achieve these goals, it is necessary to describe the development processes being studied. Process modeling languages (PMLs) were created to fulfill this role. Process descriptions provided by PMLs define relationships among the elements of a software process, for example, that one task must be performed before another, that a particular activity produces a particular artifact, how artifacts relate to each other in creating hierarchies of artifacts, and so on. By doing so, PMLs describe dependencies in the software development process – dependencies among artifacts and/or dependencies between tasks that occur during the enactment of the software process. This insight has been explored by development tools aiming to provide semi-automatic collection of traceability links (Spanoudakis and Zisman 2004).

Configuration management tools also provide process support. Some authors even argue that CM is one of the few software engineering domains in which process support has proven to be successful (Estublier 2001; Estublier, Leblang et al. 2005). In this case, the process support available in CM tools is limited to change control and problem reporting, which are described using state transition diagrams. Support for other parts of the development process has not been widely accepted by users, many of whom found it too daunting and difficult (Estublier, Leblang et al. 2005).
Workflow management systems are very similar to software process environments, but they are more generic and aim to model any business processes instead of software development processes. Most workflow management systems are task-based and also describe the artifacts being produced by these activities (Barthelmess 2003). An exception is the work of Lamarca and colleagues (1999), who support document-centered collaboration. This is achieved by associating active properties to the artifacts. These properties contain the knowledge about the dependencies between the tasks of the process being modeled.

Finally, Kraut and Streeter (1995) conducted a survey in a large telecommunication company to analyze the role of formal and informal communication mechanisms in the coordination of software development projects. They found that organizational task dependence – the extent to which one person’s task is related to tasks performed by people in other divisions of the company – influences project certainty and therefore the degree of the coordination of the project.

### 3.7. Design Structure Matrix in Management Science

Management scientists have also studied dependencies between artifacts and dependencies between tasks using an approach known as the design structure matrix (or DSM, for short). This approach is described below.

#### 3.7.1. Introduction

The design structure matrix is a tool that displays the relationships between elements of a system in a compact, visual, and analytically advantageous format
Initially proposed by Steward (1981) to describe relationships in product development processes, DSMs have been used to analyze complex systems such as organizations, products, and processes. To model such diverse uses of DSM, Browning (2001) has classified DSM in four main types:

- Component-based DSMs are used to model system architectures, or products based on components and/or their interdependencies;
- Team-based DSMs model organizational structures according to people and/or groups;
- Activity-based DSMs model processes and networks of activities or tasks and the information flow among those activities; and
- Parameter-based DSMs model low-level relationships between design decisions and parameters, systems of equations, subroutine parameter exchanges, and so forth.

These four types of DSMs are aggregated into two main categories: static and time-based DSMs (Browning 2001). The distinction between them is based on the simultaneous existence of the elements. In static matrices, the elements of the DSM exist simultaneously. In time-based matrices, the ordering of rows and columns indicates a flow through time so that some elements precede others: an element can exist only after the preceding element ceases to exist. Component and team-based DSMs are static and therefore analyzed with clustering algorithms. Activity and parameter-based DSMs are time-based and typically analyzed using sequencing algorithms. Despite these differences, concepts underlying the DSM approach are very similar, as described below.

For the purposes of this dissertation, two types of matrices are interesting: component-based and activity-based. According to Browning’s review of the literature, these models influence each other: the structure of the product influences its development process (Baldwin and Clark 2000; Nightingale 2000).
and, conversely, a legacy development process might constrain the structure of products, not allowing them to use architectures that would allow innovation. Before defining these two types of matrices, I define DSM next.

Basically, a DSM is a square matrix where rows (and columns) describe the elements that compose a complex system. A mark X in row i, column j indicates that element i interacts with or is dependent on element j. The diagonal of the matrix is not used. It is possible to replace the marks (X’s) in the matrix by values quantifying the strength or degree of the dependence between two elements.

In both static and time-based matrices, the analysis is based on reordering the rows and columns of the matrix. DSMs constitute a powerful approach that allows one to analyze both product and task dependencies in similar ways.

### 3.7.2. Component-Based DSMs

In general, in component-based DSMs, the elements of the matrix that are used in the rows and columns are the modules that compose the product being analyzed, for instance, the components of a car, computer, airplane engine, or software application. These DSMs are analyzed by using clustering algorithms that aim to find “subsets of DSM elements that are mutually exclusive or minimally interacting” (Yassine 2004). The ultimate goal is to identify modules of a system that are as independent as possible.

Baldwin and Clark (2000) adapted DSMs to make use of the information-hiding principle (Parnas 1972). More specifically, components of a DSM are separated into their interfaces and implementation parts. Interfaces are

---

12 Interfaces are called design rules, and implementation parts are called hidden parameters.
represented separately in these adapted DSMs to indicate their fundamental roles in handling dependencies. The resulting design structure matrix can then be clustered to generate modular designs.

Component-based design-structure matrices have also been applied to software design. Sullivan et al. (2001) apply them in software design and extend them to model environment variables in which the software is intended to be used. Their goal is to be able to account changes in the environment and their implications in the software design.

3.7.3. Activity-Based DSMs

An activity-based DSM is a matrix where rows (and columns) describe the tasks or activities that compose a particular development process of an organization\textsuperscript{13}, for example, Intel’s semi-conductor development process described by Eppinger (2001). Activity-based DSMs can be applied to organize the design of a system, to develop an effective engineering plan, to show where estimates are required, and to analyze the flow of information that occurs during the design work (Steward 1981). Activity-based DSMs are especially useful in concurrent engineering processes, where tasks are performed in parallel. Furthermore, activity-based DSMs allow the easy identification of interdependent tasks, which indicate the need for iterations in the process being modeled.

Similar to component-based DSMs, activity-based DSMs can be reorganized by rearranging the matrix to minimize iterations or to include tasks in the planned iterations. Other possibilities include having team members participate

\textsuperscript{13} Identifying the sequence of steps of a process is left out of the scope of the DSM. One might use interviews, surveys, and/or organizational documents to identify this process.
in coupled tasks, introducing new tasks in the process to simplify subsequent iterations, and redefining tasks within coupled groups (Eppinger 2001).

Eppinger and his students have used DSM matrices to compare, improve and extend organizational processes of product development in different companies. For instance, Morelli, Eppinger, and colleagues (1995) studied a project in the manufacture of electrical technologies. Mainly, they wanted to find out if it is possible to predict technical communication in a project. In order to do this, they created a task-based DSM of the product development process by interviewing key informants. This task-based DSM was mapped into a team-based DSM. The idea was that by representing the dependencies between the tasks, this DSM also represents the communication paths that need to take place to handle these dependencies; therefore, this matrix could be used to predict technical communication in the project. They then collected weekly questionnaires with information regarding the communication frequency of members of the teams involved in the product development. After the aggregation of these questionnaires, they compared this resulting matrix of actual communication with the one of predicted communication. Among the results of the comparison, they concluded that a majority (81.1%) of the interactions was predicted. More specifically, nearly all of the frequent and most of the occasional communications were predicted. These results suggest that team members dealing with interdependent tasks should not have communication barriers between them, since they need to interact to handle their dependencies. This method can be used to allocate team members in distributed projects, for instance.
3.7.4. Combined DSMs

Steven Eppinger and his students at MIT have conducted different studies to explore the relationship between product architecture and organizational structure – that is, how the architecture of a product (its components and associated dependencies) relate to the organizational structure (the division of labor in teams and their interactions). These studies are described in this section.

Sosa, Eppinger, and colleagues (2002) conducted a study of a project that involved both software and hardware development in the telecommunications industry. They wanted to determine the influence of some factors in the frequency of technical communication. Their results indicate that communication frequency correlates positively with the importance of the dependency relationship and organizational bonds, but decreases with distance. This holds across all media studied, suggesting that “apparently, people involved in critically interdependent tasks or who share strong organizational bonds engage in a broad spectrum of communication means.” Even when team members were non-collocated, higher communication frequencies were observed for highly interdependent pairs when compared to non-collocated independent pairs. These results reinforce the importance for managers to identify critical task dependencies in their organizations to facilitate intense communication among the team members involved in such tasks. In addition, the authors argue that by documenting communication frequencies, managers can uncover the underlying structure of products, or, more important, unidentified dependencies. To quote the authors: “tracking electronic-based communication frequencies can provide
an easy and non-disruptive way to obtain the dependency structure of a development project.”

In a different study, Sosa et al. (2003) compared two different DSMs during the design of a commercial aircraft engine. The study focused on understanding teams’ interactions: they wanted to compare the interaction of design teams that develop modular systems and the interaction of design teams that develop integrative systems. Initially, they created a component-based DSM by collecting information from key informants regarding the several systems’ interactions in the product. They then collected information about the frequency and importance of technical interactions among the teams in order to generate a team-based DSM. Finally, they compared both matrices, aiming to identify matched and unmatched interactions\(^\text{14}\). Matched and unmatched interactions and design interfaces are defined in Figure 2.

<table>
<thead>
<tr>
<th>Team Interactions</th>
<th>Unmatched design interfaces</th>
<th>Aligned absence of interfaces and interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>Aligned presence of interfaces and interactions</td>
<td>Unmatched team interactions</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Figure 2 - Mapping Design Interfaces and Team Interactions**

When the component and team-based DSMs were compared, about 90% of the cells matched. They were cells with either known product dependencies that were matched by team interactions or cells with no product dependencies and no corresponding reported team interactions.

\(^{14}\) Note that this assumes a one-to-one relationship between product subsystems and teams; that is, for each subsystem there is only one team designing it.
3.8. Conclusions

In this chapter, I presented two different perspectives to the study of dependencies that can be found in the literature.

First, dependencies between software artifacts have been studied mostly by software engineers. Their ultimate goal has been to improve the quality and maintainability of a software system and allow software artifacts to be reused. Management researchers have also looked at dependencies between artifacts through the usage of design structure matrices.

Second, dependencies between tasks is an approach based on the seminal work proposed by Thompson (1967) and adopted by organizational theorists. Malone and Crowston (1994) argue that dependencies between artifacts emerge as dependencies between the tasks required to manipulate these artifacts. Staudenmayer (1997) adopts a similar approach and describes team strategies used by software developers to handle both dependencies between software components and between tasks.

This relationship between tasks and products is exactly what Parnas (1972) and Conway (1968) proposed and which was observed in recent empirical studies (Morelli, Eppinger et al. 1995; Grinter 2003; de Souza, Redmiles et al. 2004). Management scientists have recently started to study the relationship between dependencies in products and tasks (Morelli, Eppinger et al. 1995; Sosa, Eppinger et al. 2004).

In addition, these results suggest that one can investigate coordination issues in software development by analyzing the dependencies among software development artifacts. That is, dependency analysis techniques can be used to
understand and facilitate software development efforts. It is this investigation that I describe in the following chapters of this dissertation.
Chapter 4 - Field Studies: Settings and Methodology

This chapter describes the details regarding the two field studies that I performed. Initially, I will discuss the methods that I used for data collection and analysis on each of these studies. This discussion will be followed by a description of the organizational settings in which these studies took place, including information about the size, location, and organization of the teams. All names were changed to pseudonyms to protect the privacy of the informants.

4.1. Methodology

4.1.1. Introduction

As discussed in the second chapter, I wanted to identify software developers’ individual strategies. In order to do that, I needed to adopt qualitative methods of investigation. In other words, it is necessary to adopt a qualitative perspective for data collection and analysis. This perspective allows me to collect information about what strategies are used, and, more important, how and why they are used.

4.1.2. Site Selection

To answer the research questions proposed, I chose sites where informants were software developers engaged in one or more development projects. To get a broader understanding of the phenomena of interest, it was important to choose informants who played diverse roles in this effort, from user-interface experts to
quality assurance personnel. This translated into the need to study software development efforts in different phases of the software development process. Similarly, it was important to collect information about projects conducted in different organizations and developing different types of software products. Having informants who play different roles in different software development projects that take place in different organizations is necessary to compare and contrast the data collected, and consequently to refine the phenomena being studied, improving the quality of the outcomes of this research endeavor.

Software engineers with the desired profiles were asked to participate on a voluntary basis. This means that I was not able to study one particular software development team but had to collect data about a subset of team members only.

4.1.3. Data Collection Methods

The methods for data collection adopted in this research are two methods commonly used with qualitative data collection: observation and interviews. Both are explained in this section.

Observation

Observation of informants’ activities was adopted to document their everyday practices. The main idea of this approach is to observe developers in their workplace performing the activities that they usually do in their daily jobs. To be more specific, observation consists of writing notes about the informants’ activities, events, interactions, tool usage, and any other phenomena. To collect this information, I sat distant enough to not disturb the informant but close enough to be able to observe the contents of physical or digital objects that the
informant was handling. Information collected during observation was recorded without distracting the informants. Whenever possible, I accompanied the informants to meetings or any events outside their offices or cubicles. In addition, I asked clarification questions during the observations, but not without ensuring that such questions would not disturb the informants’ activities. If it was not possible to ask clarification questions during the observations, those questions were asked during interviews.

Inevitably during the data collection, the fieldworker needs to decide the level of detail to use to describe the observations (Randall and Rouncefield 2004, p. 64). For instance, Heath and Luff (1992) provided a detailed analysis that includes gesture, eye movements, and precise conversation analysis. On a more abstract level, Gonzales and Mark (2004) recorded the time their informants spent in each activity they performed. In both cases, the research questions these authors wanted to address required such a level of detail. In my case, this detail was not necessary, so I recorded information about the informants without necessarily recording time spent on activities, their gestures, and so forth.

Observation was an important method to be used in this research because it allowed me to collect information about what tools the informants used and how they used them. Moreover, it was a source of valuable insights to be explored during the interviews, which are discussed in the following section.

**Semi-Structured Interviews**

Semi-structured interviews were conducted with the intent of understand the informants’ work practices, interactions, and any other relevant phenomena. Interviews were conducted by using techniques inspired by McCracken (1988):
questions were designed to encourage the participants to talk about their everyday work, including work processes, collaboration and coordination efforts, problems, tools, and so on. Interviews were conducted at a time convenient to the employees. Whenever possible, I tried to conduct the interviews in the informant’s cubicle or office, so the informant could show me tools, documents, or any other relevant item used to perform the work. In order to guarantee that the same topics of interest were addressed in different interviews, I used an interview guide. This guide was, to some extent, used in both organizations to guarantee that similar issues were addressed.

Artifacts produced during the software development process were collected whenever possible. Examples of artifacts include email, drawings, manuals, meeting invitations, problem reports, and so forth. This data collection was performed during the entire fieldwork (i.e., during interviews and/or observations). Due to the influence of the software architecture on how dependencies are handled in a software project, I also explicitly tried to collect information about the architecture of the software being developed.

4.1.4. Data Collection Process

The data collection started with a short period of observation and information gathering, enough so that I could familiarize myself with the setting, such as tools, project terminology, participants’ names and roles, history of the project, formal and informal rules, and so on. This initial period also allowed the informants to get used to having me around the workplace. After that, a first interview was conducted to collect specific information about the setting. In essence, the focus of this first interview was just to broad my understanding
about the overall project. The rationale for this approach is to be able to adequately use informants’ time in the next interviews to be conducted, so that they can focus on what they think is important instead of explaining details about the setting.

Another period of observation followed, now with the goal of gathering information about the daily work of software developers. New interviews were conducted only after I gained a more appropriate understanding of the site and started to formulate hypotheses about the work of the software developers. In other words, “the point at which questioning becomes a worthwhile enterprise is the point at which you know enough about the domain that you can begin to ask questions you know to be relevant” (Randall and Rouncefield 2004, p. 72-73).

Some individuals participated in multiple interviews during the study. In this case, there was usually a long interview and then a couple of short follow-up interviews to clarify important aspects or to address new aspects that the data analysis raised.

It is important to be careful about the time spent in the field because informants might start to take you for granted:

Subjects become aware of the fieldworker’s developing expertise over a period of time and gear their responses to what the fieldworker “knows.” One danger is that the fieldwork may become enculturated in the setting that he/she “goes native.” (Randall and Rouncefield 2004, p. 60)

This was particularly relevant because the informants could have discovered that I am also a software engineer. Although I did not plan to broadcast this information, ethics did not allow me to lie to the informants if they asked. To avoid this situation, I kept asking naive, sometimes even stupid questions, to explore much of what was tacit to the experienced member (Randall and
In short, I not only tried to avoid making assumptions about the informants’ answers, but I also tried to avoid the informants making assumptions about my knowledge.

4.1.5. Data Analysis Methods

Grounded Theory

Before the analysis of the data, interviews were transcribed and field notes were typed. Both types of data were integrated into a software tool for qualitative data analysis, MaxQDA2.\textsuperscript{15} Datasets from the two different organizations and projects were integrated into the tool.

After that, the data collected was analyzed by using grounded theory (Strauss and Corbin 1998). This technique does not require a prior theory about the data, that is, a set of hypothesis to be tested. Instead, the goal of grounded theory is precisely to generate theory grounded exclusively on the existing data. In other words, it aims to develop a theory or explanation about what is going on in the field, or more specifically, what is available in the data collected.

Grounded theory proposes three major steps. The first step is called \textit{open coding}, in which data are micro-analyzed (line-by-line) to identify categories. Categories are grouping concepts put together under a more abstract high-order concept to explain what is going on (Strauss and Corbin 1998, p. 113). Categories are created to minimize the number of elements that the researcher needs to consider.

\textsuperscript{15}http://www.maxqda.com/maxqda-eng/
Figure 3 presents an example of open coding from a screenshot of MaxQda2. It contains an excerpt of one interview presented in the main window with line numbers and time markers (indicated by brackets), and on the left the codes assigned to the portions of the document are displayed in blue. For instance, in lines 33 to 39 the respondent answered a question about the definition of APIs. Therefore, I created a category to illustrate this aspect.

Figure 3 - Example of Coding

All the interviews and field notes were coded in a similar fashion, with categories being defined at a level of detail that could allow me to understand how software developers performed dependency management.
During the next step, **axial coding**, those categories were broken into subcategories. Whereas categories stand for phenomena, subcategories answer questions about the phenomenon, such as when, where, why, who, how, and with what consequences (Strauss and Corbin 1998, p. 125). During axial coding, one identifies properties and dimensions of each one of the categories. For instance, a category identified in my data is called impact and it is used to model the work that a developer needs to perform because of the work that another developer performed. In this case, one important attribute of this category is its degree, meaning the degree (or intensity) of an impact on one person’s work, or the amount of re-work that the person will need to perform. Lines 40 to 47 in Figure 3 address this exactly: The informant is describing the degree to which changes in the API will impact his work. Note also that this example illustrates that categories can be related to other categories: APIs can cause more or less *impact* when they change. Again, the entire dataset of interviews and field notes was analyzed to identify and classify the different dimensions of each category.

Finally, during **selective coding**, the most important categories are selected to be core categories, that is, the categories that will be used to describe the emerging theory. Other categories are then associated with the core categories. In my case, impact will be one of the central categories of my theory. In contrast, APIs are less central and will be used to illustrate a technical approach used by software developers to reduce the impact on each other’s work.

To summarize, grounded theory provided me a perspective for examining software developers’ everyday work practices and understanding how the management of dependencies is incorporated into these practices. In order to do that, I closely examined the details of the interactions of the software engineers
among themselves and with their tools. These results are discussed in the next chapter, but initially I will describe the details about the two teams studied in the following section.

4.2. Organizational Settings

4.2.1. The MVP Study

The Software

The team I studied is part of a federal government agency and develops a real-time software application called MVP, which is composed of ten different tools in approximately one million lines of C and C++ source code. MVP has been developed over a nine-year period by researchers from Ames and external contractors. Some of the MVP tools are included in the official release of the MVP software, and others are still in development. Each one of the MVP tools uses a specific set of “processes.” A process for the MVP team is a program that runs with the appropriate run-time options\textsuperscript{16}. Running a tool means running the set of processes required by this tool with their appropriate run-time options.

Processes are also used to divide the work; that is, each developer is assigned to one or more processes and tends to specialize in it. There are process leaders and process developers. Both, most of the time, work only with the one process. This is an important aspect because it allows these developers to deeply understand a process’s behavior and become familiar with its structure, therefore allowing them to deal with the complexity of the code.

\textsuperscript{16} A MVP process is not formally related to the concept of processes in operating systems and/or distributed systems.
During the development activities, managers tend to assign work to developers according to these processes to facilitate the learning process. However, it is not unusual to find developers working in different processes. This might happen due to various circumstances. For example, before launching a new release, the entire workforce is needed to fix bugs in the code; therefore, developers might be assigned to fix these bugs no matter where they are located. Or, a developer who already started working on a bug, because it seemed to be located in his process, might later find out that the bug is located in a different process. In this case, it is easier to let the initial developer fix the bug due to the time already spent understanding it, than to assign it to a different developer.

The Team

The MVP team is divided into two groups: the developers and the verification and validation (V&V) staff. Developers are responsible for writing new code, bug fixing, and adding new features to the MVP software. This group is spread out into several offices across two floors in the same building and comprises 25 members, three of whom are researchers who write their own code for exploration purposes.

V&V members are responsible for testing the MVP software and reporting bugs (if necessary), keeping a running version of the software for demonstration purposes, and maintaining the documentation (user manuals) of the software. This group comprises five external contractors and the V&V leader, who is a civil servant. Half of this group is located in a room called the V&V Laboratory, and the other half is located in offices on the same floor in the building. The V&V staff deals only with the MVP tools that already have official releases, with the
exception of the configuration management manager, who provides support for all MVP tools.

Both, the V&V Lab and developers’ offices are located in the same building. Experience within the MVP group ranges anywhere between 2½ months to 9 years in this project. Experience with software development in general is even more diverse: it ranges from 2½ months to 26 years.

The Software Development Process

The MVP group adopts a formal software development process that prescribes the steps that need to be performed by the MVP team members during their activities. For example, after finishing the implementation of a change, all developers must integrate their code with the main baseline. In addition, developers are responsible for testing their own code to guarantee that when they integrate their changes, they will not insert bugs into the software, or, “break the code,” as it is informally characterized by the MVP developers.

Another part of the process prescribes that, after checking-in files in the repository, a developer must send email to the software development mailing list describing the problem report (PR) associated with the changes, the files that were changed, and the branch where the check-in was performed, among other pieces of information.

As a last example, the MVP software process prescribes the adoption of code reviews before the integration of any change into the main baseline, and design reviews for major changes in the software. Code reviews are performed by process leaders. If a change involves, for example, two processes, a developer’s code will be reviewed twice: once by each process leader. Meanwhile, design
reviews are recommended for changes that involve major reorganizations of the source code. The MVP manager decides their need usually during the pre-design meetings, the bi-weekly software developers meeting.

**The Tools**

MVP team members employ mainly two software development tools for coordinating their work: a configuration management system and a bug-tracking system. Of course, other tools are used, such as CASE tools, compilers, linkers, debuggers, and source-code editors, but the CM and bug-tracking tools are the primary means of coordination (Conradi and Westfechtel 1998; Grinter 1996; Grinter 1995). These tools are integrated so that there is a link between the PRs (in the bug-tracking system) and the respective changes in the source code (in the CM tool). Both tools are provided by one of the leader vendors in the market.

The MVP team employs several advanced features of the CM tool, such as triggers, techniques to reduce compilation time, labeling, and branching strategies. Indeed, the branching strategy employed is one of the most important aspects of a CM tool because it affects the work of any group of software developers. This strategy is a way of deciding when and why to branch, which might make the task of coordinating changes easier (Walrad and Strom 2002). According to the nomenclature proposed by Walrad and Strom (Walrad and Strom 2002), the following branching strategies are used by the MVP team: (1) branch-by-purpose, by which all bug fixes, enhancements, and other changes in the code are implemented on separated branches; (2) branch-by-project, by which branches are created for some of the development projects; and (3) branch-
by-release, by which the code branches upon a decision to release a new version of the product.

The branch-by-purpose strategy supports a high degree of parallel development but at the cost of more complex and frequent integration work (Walrad and Strom 2002). According to this strategy, each developer is responsible for integrating his or her own changes into the main baseline. This approach is often called “push integration” (Appleton, Berczuk et al. 1998). After that, the changes are available to all other developers. Therefore, if one bug is introduced, other developers will notice this problem because their work will be disrupted.

MVP developers create new private branches for each new bug fix or enhancement, whereas the CM manager creates branches for new projects and releases only. In the MVP team, the branch-by-purpose delivered its advantages. Indeed, I observed and collected reports of different instances of this situation. When one developer suspects that there is a problem introduced by recent changes, he or she will contact the author of the changes and ask the author to check the change or to provide more information about it.

Informal and formal communications were frequent because the whole team was collocated in the same building. Pre-design meetings were held every two weeks and would be attended by all MVP members. In these meetings, developers would share information about their work by reading the PR number and title of the PR they were working on. Major announcements about demonstrations, changes in tools, and external reviews would also be announced in this meeting. Non-face-to-face communication among team members happened through email and phone. Announcements, emails before check-in’s,
and code review requests, among other pieces of information, were all sent by email. In addition, whenever a PR changed state, the bug-tracking system would also generate an email and send it to all team members. This way, developers could be aware of their colleagues’ work.

Data Collection

I spent eight weeks during the summer of 2002 as a member of the MVP team. As a member, I was able to make observations and collect information about several aspects of the team. I also talked with my colleagues to learn more about their work. Additional material was collected by reading manuals of the MVP tools, manuals of the software development tools used, formal documents (such as the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports, and so on.

All MVP team members agreed with my data collection. Furthermore, some of the team members agreed to let me shadow them for a few days so that I could better learn about their functions and responsibilities. These team members belonged to different groups and played diverse roles in the MVP team: the documentation expert, some V&V members, leaders, and developers. I sampled among MVP processes and tools, as well as developers’ experience in software development in order to get a broader overview of the work being performed at the site. A subset of eight MVP team members was interviewed according to their availability. Interviews lasted between 45 and 120 minutes. I sampled the software developers to be interviewed according to the dimensions explained above in order to get a broad perspective on the management of dependencies. To summarize, the data collected consist of a set of notes that resulted from
conversations and documents, and observations are based on shadowing developers.

**The Organization of the Work in the MVP Team: Problem Reports**

Problem reports are an instantiation of the concept of change requests in configuration management systems. Basically, PRs are used to organize changes in the software system being versioned and, by doing so, they allow one to control and audit how the software evolves (Dart 1991). For instance, a bug in a software system needs to be initially reported as a PR in the bug-tracking system. This PR is then assigned to a software developer who will implement the bug fix. The CM and the bug-tracking tools are integrated, so that when a developer checks-in new changes in the code, that developer will associate these changes with the respective bug (PR). By adopting this convention, developers, managers, and testers find out which changes in the software are required to fix a particular bug. Similarly, the MVP software development process prescribes that new requirements to the MVP software need to be entered in the bug-tracking tool as PRs before any changes in the code are allowed in the repository. Whenever a new functionality is implemented, the changes in the code that implement this functionality will be associated with the PR that describes it.

The administrator of the bug-tracking tool can specify the (text) fields that compose the PRs. In the MVP project, fields include: the “problem,” which describes the software issue; “how-to-repeat,” which describes how to repeat a particular bug (by which tools, processes, and data); the “priority” of the bug; and others. Modern systems, such as the one used by the MVP team, also allow one to define the “life cycle” of a PR, that is, the set of states that a PR is required
to follow and the fields that must be filled to allow transitions from one state to another. A quick summary of these states and fields as reported by the MVP software manager is provided below:

- New state – Problem/Requirement, How-to-Repeat;
- Open;
- Assigned;
- Work-In-Progress (WIP);
- Resolved – DesignRev (documents design review activities), DesignNar (describes the design used in the PR), DevTest (how-to-test the design and code changes), CodeRev (code review activities), Resolution (the resolution of the code reviews);
- Verified – Verification-info; and
- Closed.

Note that some fields are prompted when the PR moves into Resolved, but MVP developers are encouraged to fill them out when they are in the previous state (Work-in-Progress). Furthermore, the bug-tracking tool sends emails to MVP members whenever PRs change their state.

4.2.2. The MCW Study

The project studied, called MCW, is responsible for developing a client-server application that had not yet been released during the period of the study. This application had been developed over nine months of the period of my data collection.

The Organizational Setting

The fieldwork described in this section was conducted during the summer of 2003 in a software development company that I will call BSC. BSC is one of the largest software development companies in the United States, with products ranging from operating systems to software development tools, including e-business and tailored applications.
At the time of the study, BSC had recently adopted a strategy of developing reusable software application components. This strategy aimed to create software components (each one developed by a different project) that can be used by other projects (teams) in the organization. According to this strategy, all components should be designed according to a reference architecture made available by senior software architects of the organization. This will be discussed in more detail below.

The Team

The project studied, called MCW, is responsible for developing a client-server application and had a staff with 57 software engineers, user-interface designers, software architects, and managers, divided into five different teams designated as follows: lead, client, server, infrastructure, and test. The lead team comprised the project lead, development managers, software architects, and user interface designers. The client team was developing the client side of the application, and the server team was developing the server side of it. The infrastructure team worked with components to be used by both the client and server teams, with installation mechanisms and documentation. Finally, the test team was responsible for the quality assurance of the product, testing the software produced by the other teams. Both client and server teams had software architects in addition to managers.

Experience within the MCW group ranged anywhere from 1 to 8 months. Experience with software development in general was different: a developer reported to be working in this area for 14 years or more, whereas others had just joined this project after finishing college.
The MCW client team held weekly meetings, with all developers attending these meetings because they were all collocated with the exception of one developer. During these meetings, the manager would ask for progress reports from the developers; in addition, they would pass on important information about other teams that could potentially impact their own team, as well as other tips about programming, tools, and so on.

MCW server meetings, on the other hand, were held through teleconferencing because most developers were distributed in two locations, one hour apart from each other. The MCW server architect was located in another country, but he would often visit one of the locations where the rest of the team was located.

MCW client, installation, and part of the server teams were all collocated.

The Tools

MCW developers used different tools in their daily work. Among the software development tools, it is possible to include a flexible open source IDE and a configuration management tool. Indeed, the MCW team had recently changed from CVS\(^\text{17}\) to a CM tool provided by a leader in the market. This was a major cause of concern and frustration among software developers because they had major problems adapting to the new tool. All other teams involved in the organization-wide reuse program had already migrated to this new CM tool.

Storage of requirements documents and bug-tracking information were accomplished by using discussion databases in Lotus Notes, which also was the

\(^{17}\text{http://www.nongnu.org/cvs/}\)
email client used by the team and the whole organization. MCW developers would email their colleagues about new information (e.g., new CM instructions) being posted in the discussion databases.

In addition to email and discussion databases, developers relied heavily on the instant messenger (IM) system used across the entire organization: any BSC employee could be found through this IM system. They used IM to contact colleagues or managers for help. Because of the organizational usage of the IM system, a developer could contact any other developer (or employee for that matter) in the whole organization, given a developer’s name and location. MCW developers were expected to be online in the system when at work, which also meant that they could be interrupted.

Data Collection

I spent 11 weeks at the field site performing data collection using the methods that I described in the previous section. Among other documents, I collected meeting invitations, emails and instant messages exchanged among the software engineers, and product requests for software changes. I was also granted access to shared discussion databases used by the software engineers. All of this information was used in addition to the field notes generated by the observations and interviews. I conducted 15 semi-structured interviews with members of all five sub-teams. Interviews lasted between 35 and 90 minutes.

During my fieldwork, I noticed the fundamental role of the software architecture, especially of the application programming interfaces or APIs, in the coordination of the work. Therefore, I collected more information about this aspect in order to verify whether I had understood the software developers’
work. In addition, the interviewees provided feedback on my interpretation of their software development process, with particular emphasis on the roles of APIs in the coordination of their work. This feedback was fundamental to improve my understanding of their work. The next section describes in more detail the software architecture suggested by the organization and adopted by the MCW project.

Software Architecture: Layers, APIs, and Data Objects

The BSC reference architecture prescribed the adoption of some particular design patterns (Gamma, Helm et al. 1995), but at the same time it gave software architects across the organization flexibility in their designs. It is based on tiers (or layers) so that components in one tier can request services only to the components in the tier immediately below them (Rumbaugh, Blaha et al. 1991) (Buschmann, Meunier et al. 1996). According to the BSC reference architecture, a layered architecture minimizes the coupling between the tiers and consequently allows developers to work in parallel instead of sequentially:

“On large projects, we need to rely on the ability of [U] tier and [S] tier developers to work independently. … [special objects] can be defined to allow user tier programmers to build and test their code with a ‘stubbed’ implementation before the services have been written.”

Data exchange between tiers is possible through well-defined objects called “value objects.” Meanwhile, service requests between tiers are possible through APIs that hide the details of how those services are performed (e.g., either remotely or locally, with cached data or not, etc.). In other words, software components should have a public and stable API through which their consumers can access the set of services provided by that component. In this organization, APIs were designed by software architects in a technical process that involved
the definition of classes, method signatures, and other programming language concepts, and the associated documentation.

The reference architecture contained recommendations about how to design “special objects” for data exchange between tiers. However, it did not contain recommendations about how to design the APIs. Indeed, during one of the meetings that I observed, developers from different projects discussed the lack of recommendations from software architects on how to proceed when designing APIs. As one developer pointed out: “All APIs need to look, feel, and smell the same.” This lack of an established process had already been identified by the software architects who were starting to discuss it in the software architects’ weekly meetings. That caused some problems in the development process, as I will discuss in the next chapter.

Application Programming Interfaces

In order to understand the concept of application programming interfaces, it is necessary to understand a couple of important software engineering principles first. Separation of concerns, for example, is one the most important principles in software engineering. It allows one to deal with different individual aspects of a problem so that it is possible to concentrate on each separately. When different parts of the same system are dealt with separately, this a type of separation of concerns named modularity (Ghezzi, Jazayeri et al. 2003). Modules should be designed according to the information-hiding principle proposed by Parnas (Parnas 1972). According to this principle, software modules should be both “open (for extension and adaptation) and closed (to avoid modifications that
Affect clients)” (Larman 2001). An API is defined by the Software Engineering Institute as:

Application Programming Interface (API) is an older technology that facilitates exchanging messages or data between two or more different software applications. API is the virtual interface between two interworking software functions, such as a word processor and a spreadsheet. ... An API is the software that is used to support system-level integration of multiple commercial-off-the-shelf software products or newly-developed software into existing or new applications. (2003)

Although this definition presents APIs as interfaces between software applications, among professional software engineers the term API is coming to mean any well-defined interface that defines the service that one component, module, or application provides to other software elements. Therefore, I will adopt a looser definition of an API as proposed by des Rivieres (2004b): An API is a well-defined interface that allows one software component to access programmatically another component and is normally supported by the constructs of programming languages. Typically, in a programming language such as Java, an API corresponds to a set of public methods of classes and interfaces and the associated documentation (in this case, javadoc files). In the rest of the text, I will use the terms “component,” “module,” and “software applications” indistinctly because they do not change the purpose of using APIs.

The word “interface” in “application programming interface” is used to explicitly indicate that APIs are constructs that exist in the boundaries of at least two different software components or applications. For instance, the Microsoft Windows API allows a program to access and use resources of the underlying operating system, such as file system, scheduling of processes, and so on. The
two (or more) applications that an API divides are often developed by different teams, and hardly ever by individuals.

APIs are largely adopted by industry because they support the separation of interface from implementation, a growing trend in software design (Fowler 2002). The main advantage of this approach is the possibility of separating modules into a public (the API itself) and a private (the implementation of the API) part, so that changes to the private part can be performed without impacting the public one, therefore minimizing the dependencies between these two parts (Parnas 1972). Indeed, even distributed software development teams adopt APIs to isolate their work (Grinter, Herbsleb et al. 1999).

In the rest of the text, I adopt the terms API consumers and API producers. API consumers are software developers who write code with method calls to an API, and API producers are software developers who write the API implementation.

An important aspect of any API is stability. A stable API is not subject to frequent changes, therefore leveraging the promised independence between the API producers’ and consumers’ codes. Changes in the API itself require changes in the API consumers’ code because this code uses services provided by the API. This situation might become problematic if changes to the API happen too often. According to a software architect interviewed, APIs “tend to be something well-thought out, and set in stone,” so they are regarded as contracts with the clients. As a result, API consumers expect that the API will not change often, and if it does change, they also expect that these changes will not severely affect them. Recent work in software engineering tries to provide advice on how to properly
change APIs so that the impact of those changes is minimized (des Rivieres 2004a; Fowler 2002).
Chapter 5 - Dependency Management as Impact Management

5.1. Introduction

This chapter presents the results of the two ethnographic field studies that I conducted in order to answer the research questions proposed\(^{18}\). I will describe, through the presentation of ethnographic material, how the management of dependencies is woven into the daily software development activities performed by MVP and MCW developers. In doing so, I will show how the work of dependency management can be seen as one of impact management: managing the impact of one’s effort into others and at the same time, the impact of others into one’s own effort. Impact management is accomplished by software developers through a variety of approaches, including artifacts, formal and informal practices, and with the help of groupware and software development tools. I will illustrate each one of those different aspects in the following sections. Initially, however, I will describe in more detail the concept of impact management.

5.2. Impact Management

Dependency management is the work to manage the effects of software dependencies in the coordination of the work. As I will describe, dependency management can be seen as impact management. Impact management is the

---

\(^{18}\) Details about the settings where the studies were performed can be found in the previous chapter.
work performed by software developers to minimize the impact of one’s effort on others and, at the same time, the impact of others into one’s own effort.

Viewing dependency management as impact management draws attention to the software developers’ concern about impacting or being impacted by colleagues in development efforts. It illustrates how one orients himself toward his colleagues so that both can get their work accomplished at the end of the day.

There are three main aspects in impact management:

1. First, finding the network of people that might affect one’s work and that might be affected by one’s work, the impact network. This network is influenced by several factors, but as I will discuss, mainly by the software architecture\(^\text{19}\). Identifying the impact network is the most important aspect of impact management, and in fact my data suggest that when the impact network cannot be properly established, software development becomes more complex.

2. Second, forward impact management is the work to assess the impact of one’s own work on one’s respective impact network and inform them of such impact. Examples of these practices include filling the fields of the problem reports in the bug-tracking tools, sending emails with impact descriptions, code reviews that aim to understand the impact of the changes in the software architecture, and so on.

3. Finally, backward impact management consists of assessing the impact of the work performed by developers in one’s impact network on one’s own

\(^{19}\) I am using the term software architecture to loosely indicate how the software is organized into parts and how these parts are interconnected.
work, and the appropriate actions to avoid such impact. Software developers will adopt several approaches to avoid such impact, such as “back merges” that incorporate changes from others into one’s workspace, attending meetings with other teams, and so on. As expected, as much as possible, software developers try to anticipate changes in the software system so that they can prepare themselves for these changes.

Note that the distinction among these aspects is analytical: they are iterative, interwoven among themselves and among other software developers’ practices. Furthermore, these aspects are complementary: whenever a developer is performing forward management, he is facilitating the backward management to be done by others, and vice versa. For instance, whenever a MVP developer decides to postpone his or her check-in’s (forward management), he or she is allowing other developers not to worry about deciding whether to recompile their code (backward management). Note also that the analytical focus is on the strategies of the MVP and MCW teams. Whenever a MCW team member decides to attend meetings of another team that the MCW team member is dependent upon, he or she is doing backward management. This team member is doing extra work to understand how their changes can have an effect on him. For the purposes of this dissertation, however, I am not categorizing the other team’s work of allowing an external developer to attend a meeting as forward impact management. That being said, some strategies to be described embody forward and backward

A careful reader might notice a parallel with two complementary aspects of awareness in CSCW: on the one hand, actors monitor the activities of their colleagues and demonstrate such monitoring, and on the other hand, actors make their own activities publicly visible to others (Heath and Luff 1992; Schmidt 2002). I will discuss this in detail in Chapter 8.
management concerns; they are complementary aspects of the same coordinative practices. Other strategies are specific (either backward or forward) and do not have a counterpart within the MCW or MVP team.

Notice that impact management strategies can be understood at different levels of analysis: Some practices are performed by individuals, whereas others are adopted by the team as a whole (often suggested by the manager), and finally some of them are organizational mandates. Table 1 presents a summary of the strategies that I identified with information about the unit level applied to them.

Organizational strategies are those mandated by the organization, namely, the implementation of the reference architecture and the usage of application programming interfaces. Team strategies are those mandated by managers, usually described in the software development process. Finally, individual strategies are those mundane practices performed by software developers, the “ordinary methods that ordinary people use to realize their ordinary actions” (Coulon 1995, p. 29).

In the following sections, I will describe the different aspects of impact management by presenting the practices associated with each team and performed by MVP and MCW developers.
### Table 1 - Summary of Strategies

<table>
<thead>
<tr>
<th>Team</th>
<th>Strategy</th>
<th>Aspect</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVP</td>
<td>Learning from Email Notifications</td>
<td>Impact Network</td>
<td>Individual</td>
</tr>
<tr>
<td>MCW</td>
<td>API Design Review Meetings</td>
<td>Impact Network</td>
<td>Team</td>
</tr>
<tr>
<td>MCW</td>
<td>Personal Network</td>
<td>Impact Network</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Sending Email Notifications</td>
<td>Forward</td>
<td>Team</td>
</tr>
<tr>
<td>MVP</td>
<td>Reading Email Notifications</td>
<td>Backward</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Impact Descriptions</td>
<td>Forward</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Error-Checking</td>
<td>Backward and Forward</td>
<td>Individual</td>
</tr>
<tr>
<td>MCW</td>
<td>The Reference Architecture and APIs</td>
<td>Backward and Forward</td>
<td>Organizational</td>
</tr>
<tr>
<td>MCW</td>
<td>Pre-Testing Activities</td>
<td>Backward and Forward</td>
<td>Team</td>
</tr>
<tr>
<td>MCW</td>
<td>Build Document</td>
<td>Forward</td>
<td>Team</td>
</tr>
<tr>
<td>MVP</td>
<td>Back Merges</td>
<td>Backward</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Partial Check-in’s</td>
<td>Backward</td>
<td>Individual</td>
</tr>
<tr>
<td>MCW</td>
<td>Handling Dependencies – APIs and adaptors</td>
<td>Backward</td>
<td>Organizational</td>
</tr>
<tr>
<td>MCW</td>
<td>Handling Dependencies – “Exporting” developers</td>
<td>Backward</td>
<td>Team</td>
</tr>
<tr>
<td>MCW</td>
<td>Handling Dependencies – Being aware by attending meetings, engaging in communication, and so on.</td>
<td>Backward</td>
<td>Individual</td>
</tr>
<tr>
<td>MCW</td>
<td>Handling Dependencies – Group requirements</td>
<td>Backward</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Problem Reports</td>
<td>Forward</td>
<td>Team</td>
</tr>
<tr>
<td>MVP</td>
<td>Formal Code Reviews</td>
<td>Forward</td>
<td>Team</td>
</tr>
<tr>
<td>MVP</td>
<td>Informal Code Reviews</td>
<td>Forward</td>
<td>Individual</td>
</tr>
<tr>
<td>MVP</td>
<td>Holding onto Check-in’s</td>
<td>Forward</td>
<td>Individual</td>
</tr>
<tr>
<td>MCW</td>
<td>Notifications</td>
<td>Forward</td>
<td>Individual</td>
</tr>
</tbody>
</table>

### 5.3. Impact Network Identification

The impact network of a software developer is the set of developers that might affect this developer’s work or that might be affected by this developer’s work. MVP and MCW software developers faced very different situations while identifying their impact networks.
5.3.1. MVP

MVP developers did not face problems identifying their impact network due to several factors. Most developers had been working in the project for a couple of years and had learned about which developers work in which parts of the source code. In addition, the software development process prescribed that software developers should send email to all other developers before checking-in code. These emails are broadcast to the entire set of software developers and V&V personnel, who then uses this information to find out whether the new check-in affects their work or not. Especially because of this step in the MVP process, developers did not need to worry about identifying their impact network: they would always inform and be informed about changes in the code.

Broadcasting emails was also useful because of the lack of modularity of the MVP software: a change in one particular “process” could impact all other “processes.” According to the most senior MVP developer:

*MVP-developer-08:* “There are a lot of unstated design rules about what goes where and how you implement a new functionality, and whether it should be in the adaptation data or in the software, or should it be in [process1] or should it be in [process2]. Sometimes you can almost put functions anywhere. Every process knows about everything, so just by makefiles and stuff you can start to move files where they shouldn’t be, and over time it would just become completely unmaintainable.” [Emphasis added]

*Interviewer:* “I remember that a developer was telling me that there is a bunch of files, like [files] or something, and everyone works there … any time some people change something there, you have to recompile everything.”

*MVP-developer-08:* “Yeah. It’s because they are used by multiple processes. I don’t know how to get around that – maybe have smaller subsets or something, but … yeah, every process talks to every other one in a way and there are some basic architecture issues …” [Emphasis added]

In fact, one of the “process” leaders reported that his MVP team, given the current dependencies of its software, should probably be looking into refactoring tools to reorganize the software in order to reduce its dependencies. Overall, this suggests that the impact network could be as large as the entire team of
developers, which would make it necessary to broadcast changes to all developers.

However, MVP team members sometimes needed more contextualized information about their impact network. This was accomplished by using information available on these emails. They associated the author of the emails describing the changes with the “process” where the changes were occurring. In other words, MVP team members assume that if one developer repeatedly performs check-ins in a specific process, it is very likely that he is an expert on that process. Therefore, if another developer needs help with that process he will know who to contact for help. For instance, according to MVP-developer-04:

“If you are used to looking at the headlines and know that [tool1] stuff seems to always have [MVP developer1]’s name on it and all of a sudden you get a bug, for us with the GUI because you can get it from any point, I could end up with a GUI bug that ends up being [tool1]-ish in the PGUI and what do I do? I don’t understand why this thing behaves the way it does but most of those PRs seem to have [MVP developer1]’s name on them. So you go down and see [MVP developer1] so by just reading the headline and who does what you kind of get a feeling of who does what, which isn’t always bad.

…

[MVP developers2] does [tools2] sort of stuff and although I have never had to talk to him about it, but if I run into a problem by reading the email or seeing them he tends to deal with that kind of stuff so they [the broadcast email messages] tend to be helpful in that aspect as well. If you have been around 10 years you don’t care, you already know this. I have only been here two years and that stuff can make a difference— who you ask the question to when you get in trouble.”

As he pointed out, this developer had been working for only two years in the MVP project. Knowledge about which developer works in each “process” is important for him and other newcomers because all MVP developers might need to make changes in “processes” in which they are not experts. In this case, MVP developers will often contact the process owner, trying to find out whether there

---

21 A not unusual situation, as discussed in the previous chapter.
is a problem in the process. If there is a problem, developers will start working to find a solution to this problem. Later, MVP developers will again contact the owners of the process to verify whether their changes in the process are going to impact the work of these process owners. This will be discussed as one of the forward impact management practices more specifically under the label informal code reviews.

5.3.2. MCW

In contrast to MVP developers, MCW developers faced several problems while identifying their impact network. In fact, during interviews with MCW developers, I found out that many of them were not aware of their network: MCW server developers did not know who was consuming the services provided by their components, and MCW client developers did not know who was implementing the component that they depended on. Because they were not aware of their impact network, developers did not receive important notifications (e.g., important meetings they needed to attend).

To deal with the problem of finding their impact networks, developers relied on their personal social network. A developer, for instance, reported talking to up to fifteen people before finding the right person:

Interviewer: “So have you experienced this problem?”

---

22 Sometimes bugs are reported because of an abnormal behavior that might be considered a problem; the role of the developers in this case is exactly to find out whether there is a problem. This happens because of the complexity of the MVP code that exists for several years and the lack of domain knowledge of MVP software developers (Curtis, Krasner et al. 1988). In this case, information about the problem is exchanged by using emails, and a PR is not inserted in the bug-tracking tool until the existence of a bug is confirmed.
MCW Developer-15: “Totally. That is what I have said. I am kind of merciless in trying to find the right person. I have shotgunned up to four or five people at once to say ‘do you know who is responsible for this?’ and then gotten some leads and followed up on those leads and talked to as many as ten to fifteen different people.”

Managers also played an important role in this process because of their larger social network. MCW developers would contact their managers so that these managers could identify the person they wanted to find.

Another way of bringing MCW developers together was the API design review meetings. Within the MCW team, these meetings were scheduled to discuss the APIs being developed by the server team. The following people were invited: API consumers, API producers, and the test team that eventually would test the software component’s functionality through this API. In addition to guaranteeing that the API met the requirements of the client team and that this team understood how to use it, this meeting also allowed software developers to meet. After that, the server team provided APIs to the client team with “dummy implementations” to temporarily reduce communication needs between them, thus allowing independent work. Unfortunately, time would pass between these meetings and the actual implementation of the API. In the meantime, changes in developers’ assignments would cause communication problems because developers did not know about each other anymore. In short, changes in assignments changed the impact network, thereby making the work of software developers more difficult to be coordinated.

Software developers acknowledged that this situation was problematic. For instance, a developer suggested that a database containing information about who was doing what in the BSC organization was necessary: “sometimes you wanna talk to a developer … the developer in the team who is working in this
feature [that you need].” Architects and managers also recognized this situation as problematic:

MCW Architect: “Another thing that I don’t know if people are saying but in our team meeting yesterday and in other ones, people seem to be somewhat reluctant to talk to their counterparts too much, in the sense they feel they are bugging another person and that is a problem because the reason we are here and getting paid is to get this done and if this interaction is going to happen we need to figure out how this is going to happen.”

Interviewer: “It is not common, but I have seen one or two people who didn’t know who were their counterparts.”

MCW Architect: “That is not good.”

Interviewer: “Actually, I found that on both sides. On the server team, they are providing code to somebody but they did not know to whom they are providing code.”

MCW Architect: “The problem with that too is that there is another case where people are thinking that there is someone else doing something [but] when push comes to shove and it gets pushed on to you, it is an empty void because they don’t stand up and say that they have tried to identify their server counterpart and my client counterpart and there is not one. We have a problem here.”

Identifying the impact network was a problem in the entire BSC organization. Indeed, BSC managers created yet another discussion database that developers could use to find out who were the people necessary to answer their questions. Because of the large number of databases already in use by the BSC teams, managers had to slowly convince BSC developers of the importance of this particular database:

“The management team is really trying to socialize the idea that that [the discussion database] is the place to go when you have a question and you don’t know who can answer it. They are really trying to socialize that people should give a scan to it every once in a while to see if they can help and answer a question. The amount of traffic there has picked up quite a bit in the last couple of months, especially in the past couple of weeks. My team has not gotten that message a hundred percent yet. There is a tool for it and a place to go that I have had a lot of success with when I use it; it is just that the message has not gotten out yet that that is the place to go. One of the things that happens when you have so many databases [is] it takes a while for one to emerge as the place to be. This is turning out to be the place to be.”

Similar to the MVP team, the software architecture also influenced the process of impact network identification. Applications developed in the organization should be designed according to a reference architecture based on layers, so that
components in one layer could request services only to components in the layers immediately below them. In other words, each layer had specific dependencies; if it was changed, it would be necessary to inform only the developers dependent on this layer, instead of broadcasting changes to all developers. In fact, interviewees clearly suggested to be interested in changes only in particular parts of the architecture, parts that they depended on.

Another important factor that influenced the impact network identification was an organizational one, the large-scale reuse program adopted by BSC. This program, as discussed before, led MCW developers to interact with developers in different teams who could be located in other parts of the country and even in other countries. This was necessary to allow components to be reused in the organization. In fact, a developer complained about the need to simplify the “communication channels” in the organization to avoid having to interact with different managers to find out who was the person responsible for implementing a particular component or its services. This same developer reported that one of the teams providing a component to his team was not even aware of his team’s need. On another occasion, a developer tried to find out whether she could use a particular user-interface (UI) component. The UI designer working with her indicated a developer in Japan who was using this same component. It was this Japanese developer who recommended to her another developer, now in the U.S., who was implementing the UI component she wanted!

This problem was aggravated by the young age of the project, according to [MCW Developer-15]:

“When you sit on a team for two years you know who everybody is. Even peripherally you know who people are. So if we had to get answers about [another BSC product in the market for years] we have so many people on the team who were on the team for so long [a]
period of time they can get the answer immediately. They know who the person is even if they have never met them. We don't have that in this group because it takes time for those relationships to develop, for there to be ... enough of the investigations ... like I talked to so and so and talked to so and so and so on. You only have to go through that once or twice because once you have gone through that you know the person. I think part of that frustration is how you spin up those relationships more quickly. I don't know if you realize this but this team has only been in existence since last year. So it is a ten-month-old team.”

5.3.3. Summary

Finding the impact network was most of the time not an issue for MVP developers because the software development process that they adopted required them to send email to all developers in the team. In addition, these developers were collocated and met frequently. However, novice developers expressed the need for identifying experts in the code to properly perform impact management. They accomplished this identification by using these same email messages.

At BSC, the situation was completely different. Finding people was a major problem recognized by both software developers and managers. The main reasons for that were the large-scale organizational reuse program adopted by BSC (that required developers to interact with distributed colleagues), changes in developers’ assignments, and finally the young age of the project. Despite these problems, MCW developers could still work together because of the other practices they employed to manage impact. These practices are described in the following sections.

5.4. Backward and Forward Impact Management

As discussed before, some practices adopted by developers at MCW and MVP were used for both forward and backward impact management.
5.4.1. Email Notifications by the MVP Team

Impact Descriptions as Forward Impact Management

The MVP process prescribes that after checking-in code in the repository, a developer needs to send an email about the new changes to the software developers’ mailing list. However, I found out that MVP developers send this email before their check-in’s. In addition, MVP developers add to this email a brief description of the impact that their work (changes) will have on others’ work. The following list presents some comments sent by MVP developers:

“No one should notice.”
“[description of the change]: only [tool1] users will notice any change.”
“Will be removing the following files. No effect on recompiling.”
“Also, if you recompile your views today you will need to start your own [process1] daemon to run with live data.”
“The changes only affect [y] mode so you shouldn’t notice anything.”
“If you are planning on recompiling your view this evening and running a MVP tool with live [type of data] data you will need to run your own [process1] daemon.”

Details of this approach, however, were left to be decided by the developer sending email. According to MVP Developer-02:

“sometimes when, if the thing is going to have wide impact then its up to the person who wrote that email to say that this is going to have wide impact and everyone is going to have to recompile. Like if [developer’s name] changes the format of the [process] file, something that needs to be read in, he upgrades the [component] or something like that. That means that old files may no longer be useful and the files are going to have to recompile – people have collections of these files that are going to have to be converted somehow.”

Backward Impact Management

Emails sent by software developers and generated by the bug-tracking tool are used by developers to assess the impact that the changes to be checked-in are going to have in their own work and to prepare themselves to these changes. The following excerpt from my field notes illustrates this situation:
“[MVP-developer-02] is a software developer in the MVP team. He has been away for a few days. He reads/skims the emails generated by [the bug-tracking tool] in order to check if the changes of the PR that other developers are checking-in in the main repository impact his own work. He also thinks that [reading these emails] is one way of 'being aware of what’s going on in the system.' “

The other example below illustrates how software developers use the information in these emails to decide what to do next:

“According to [MVP-developer-04], this information on the email is important to him because he knows that he shouldn’t do a full clean because [the new changes are] gonna break his code, and then to fix it, he’ll have to do back merges and he’s gonna spend 45 to 60 minutes to do it. So, if he doesn’t want to spent that time doing that, he just doesn’t do the full clean, which means that he’ll gonna be compiling his file with the previous versions of the other files that were checked-in, and consequently, his changes will still be working.”

[MVP-Field Notes]

In the example above, MVP-developer-04 was able to assess the impact that a particular change was going to have in his work, and as a result he decided to avoid incorporating this change into his workspace and continued working undisturbed. If he had chosen to incorporate the change, he would use the configuration management tool to do it. That would be followed by a 45- to 60-minute process of recompilation. Most of the time, developers will grab the latest changes to avoid working with outdated code. In fact, managers at MVP suggest that this should be done often because it allows developers to find out about possible unexpected interactions between their work and the new changes, it avoids one working with outdated code, and it reduces disk space constraints.

Emails generated automatically by the bug-tracking tool also allowed developers to anticipate their colleagues’ actions. This was possible because the bug-tracking tool sent emails when a PR changed its state. With that information, developers could anticipate that their co-workers would soon check-in code in the main repository. Developers relied on these emails to decide their next course of action. In fact, on one occasion a developer had to send an email to a colleague about a certain PR mentioning that nothing had been checked-in. The MVP
manager had changed the state of the PR by mistake, which generated an email notification.

Meanwhile, the V&V staff also monitored the email messages sent by the MVP developers and by the bug-tracking tool. They needed to do that because they were responsible for keeping an updated version of the software running in the V&V Laboratory. This version is not the main one used by all software developers, but one that is used to generate the next release of the MVP software and is located in a special branch of the configuration management tool. V&V members used the information in the emails to decide if this version of the software that exists in the release branch needs to be recompiled.

Summary

The examples described above illustrate how the emails exchanged among MVP developers are used for both forward and backward impact management. Developers add impact descriptions to their emails, therefore providing forward impact management, and MVP team members, by reading these emails, are able to understand, assess, and anticipate changes that might potentially impact their current work. With this information, developers choose carefully their next activities.

MVP developers use their knowledge of their everyday work to tune the “notification” process. They deviate from the software process and consequently simplify their colleagues’ work by not sending notifications that are not useful for some developers. This happens when developers are making changes in the new user-interface components and only send email notifications to the other user-interface developers working on this new implementation. The developers
are aware that the current MVP software is not yet dependent on this new user-interface implementation. Therefore, they do not need to inform all developers of their changes, because they cannot impact their colleagues. However, whenever developers are making changes in the old user-interface, they know that they need to email all developers because they all can be affected by their changes. In short, software developers are aware the impact network is contingent on where in the MVP software the changes are being made and consequently adjust their work practices to reflect this knowledge.

5.4.2. Error-Checking

During the interviews and observation, an aspect that emerged was the constant concern of MVP developers that their changes did not introduce errors in the software located in the main branch. Basically, two aspects need to be accounted for: they want to make sure that other check-ins in the repository do not introduce bugs in their own changes; and they want to make sure that their changes are not going to introduce bugs in the MVP software. To address these aspects, MVP developers basically grab changes checked-in in the main branch, and then recompile their source code to make sure it is still running properly. This is possible because developers are made aware of new changes checked-in the repository by reading the email notifications about changes sent before check-in’s. Furthermore, they might choose when to grab changes and recompile their code. Even if they decide not to incorporate these changes as they happen, they will have to do so in order to check-in their code in the main branch. In these cases, incorporating, recompiling, and testing the changes are performed in the developers’ private branches and in the main branch.
That MVP developers want to make sure their changes do not introduce errors is not surprising – in fact, this is even expected. What is surprising is the MVP developers’ concern with not impacting their colleagues while committing changes. The following piece from my field notes illustrates this point very clearly:

“[MVP Developer-04] starts checking-in the files. According to him, [sending email before checking-in code] as well as the full clean and full compile before this last check-in are really a ‘good thing’ because if you don’t to that, you might check-in something that breaks the [main branch], which means that ‘you’ll be screwing the workday of 16 developers because they will also be affected by your work. So, ‘you don’t make a fool of yourself’ sending three emails:

1. I am about to check-in
2. I broke something with my check-in; and
3. I fixed the problem.” [MVP Field Notes, emphasis added]

Another important point illustrated by this same quote is the developers’ concern in avoiding the public embarrassment of “breaking the code”.

Whenever MVP developers finish implementing their changes in the code, as one would expect, they will test their code to make sure these changes deliver the expected functionality. Testing at this stage is very informal: a developer will sit on the V&V Laboratory and compare the latest version of the MVP software with the one in his private branch containing his or her changes.23 MVP developers are especially careful at this point because they are concerned with impacting their colleagues and with being impacted by colleagues during the testing activities:

MVP-developer-03: “Oh yeah, everything depends on [process1] in terms of if you are trying to make a [domain-specific analysis] all the MVP processes call the [process1] for the solution. …

Interviewer: “So does it usually take longer to test …?”

23 More formal techniques, based on regression testing, for instance, are not used at this point. These techniques will be used by the V&V staff only before a new release of the software is made available.
MVP-developer-03: “Well, for me, because I like to be thorough and I don’t want to …, I find it easier for me to find the problems myself now then to just ship the thing out and have someone else find it because by the time they find it maybe you’ve forgotten what you’d done and it takes you more time to speed up and then when they find it then it is more of a hurried up situation where now it is impacting other people whereas if I tried to do my best up front and I try to cover all the bases then I might have a better idea as to what is going on if something does happen with it. So, with the [process1] I am a little more thorough …”

The above quote also illustrates the influence of the software architecture in the testing activities: a “process” with a high degree of dependencies has to be tested more carefully because it might introduce problems in all developers’ code that depends on it.

In short, MVP developers perform error-checking activities before checking new code in the main repository for two main reasons: (1) because they do not want to introduce errors in the main branch (forward management), and (2) because they want to avoid being affected by others’ changes (backward management). In order to do that, they basically perform two activities: they test their changes to make sure they deliver the expected results, and they incorporate changes committed to the repository into their own branches to find out if these changes interact with the changes they wrote.

5.4.3. The API’s Life Cycle

As discussed in the previous chapter, applications developed in BSC should be designed according to a reference architecture, which recommends the usage of APIs. Although there was no formal process establishing how APIs should be designed, an informal process was adopted by members of the MCW project. In this case, the majority of the APIs were developed by the server team, who provided services to be used by the client team. Each one of these APIs was specified by the server software architect as necessary. After an API was
specified, it was discussed by the interested parties in a formal design review meeting (see section 5.3).

Once an API was approved in the design review meeting, a first implementation was made available to its consumers through the configuration management tool. Besides the specifications of classes, methods, and interfaces, the software architect provided a shallow implementation of the API for the sole purpose of allowing the client team to immediately start programming against this API. According to one software architect: “The first-pass delivery [of an API]... is a shallow implementation, just enough to start some work.”

Software developers would refer to these dummy implementations as “local APIs,” in contrast to “remote APIs,” which are the real APIs implemented by the server team.24

Periodically, API providers replace parts of this shallow API implementation by its real implementation – which enables more and more services in the component – often based on suggestions provided by the needs of the API consumers. According to MCW server developer-04:

“when it [the implementation] is ready, I replace the dummy code for the real implementation”

By adopting the usage of local APIs, it was possible to separate the work that each team needed to perform and temporarily reduce the dependencies among them. In the MCW project, the client team could start implementing against the local API while the server team could start implementing the (real) remote API.

24 These APIs are called “remote” because when the application is released, they will be located in a remote machine. Note that “local” and “remote” APIs are examples of the same API; the unique distinction between them is their implementation.
Once independent work is supported, backward and forward impact management are also facilitated. Forward management is facilitated because changes in the internals of a component do not need to be sent to API consumers, and backward management is facilitated because API consumers do not need to engage in as much communication and coordination with the API provider. Although this independence was very useful in the beginning of the software development process, we noticed that software developers still had to engage in coordination activities. This will be discussed in later sections.

**5.4.4. The Build Document and the Pre-Testing Activities**

As discussed above, APIs are distributed to their clients with a “dummy implementation.” According to a schedule negotiated by the client and server team managers and architects, this implementation is slowly replaced by the real implementation. As new services are enabled in the component in each new build, the test team needs to be informed because they will test these new services. This is achieved through meetings among team leaders, but, more important, through the “build document” that contains the information about which new services are implemented in the build. In short, for each new build, developers update the build document that is later used by the test team to develop the test plans to be used to test this new build. These test plans are sent to developers so that they can inform the test team of possible changes in the schedule, avoiding errors that could delay the schedule of the project.

A server developer also uses this same schedule to prepare test plans to test the functionality that will be made available each week. These plans are sent to the server developers whose components are being tested and to the members of
the test team. Developers can then provide feedback on these test plans, while testers can reuse these plans when necessary. This role was created because the integration in the previous milestone took more time than expected. Therefore, the server team manager wanted to avoid problems in their code, because he knew that these problems would need to be fixed and that extra-time would impact the client team’s schedule.

In addition, because the server developers could review the test plans before they were actually performed by the testers, this allowed them to easily remove potential problems in these plans, consequently simplifying the work that they would need to perform to address these problems when the testers pointed them out. In other words, because of these pre-testing activities, the test team is less impacted by the server team’s work because when new services are enabled they are accompanied by the respective test plans used to validate these services (forward impact management). At the same time, this provides backward impact management to the server team: by simplifying the test team’s work, this developer also simplified the impact of the work of this team’s work in his team.

5.5. Backward Impact Management

5.5.1. Back Merges and Partial Check-in’s

Software developers use the emails exchanged among them to find out if they have been engaged in parallel development. Parallel development happens when more than one developer needs to make changes in the same file. This means that the same file is checked-out by different developers and all of them are making changes in the different copies of this file in their respective workspaces. If a
developer is engaged in parallel development with other developers and they checked-in their changes in the main branch before he or she did, that developer will have received emails from the other developers about their check-in’s. By reading these emails, the developer will be aware that he is engaged in parallel development because these emails need to describe, among other things, the files that have been checked-in. In this case, this developer will need to perform an operation known in the MVP team as a “back merge.” This operation is supported by the CM tool adopted by the team and is required before this developer can merge his or her code into the main branch.

Parts of the MVP software contain important definitions that are used throughout the whole program. This means that they are constantly changed by several developers in parallel; back merges thus are performed fairly often:

“It depends on … there are certain files, like if I am in [process1] and just in the [process2] that [back merges] is probably not going to happen, if I am in the [process3] there is like … there is socket related files and stuff like that. I think [filename] and things of that sort. There’s a lot of people in there. The probability of doing back merging there is a lot higher. What I will probably try to do is discard my modifications and or I’ll save my modifications and then, right now I’ll see if I can put myself on top of it because at that point there’s stuff supposedly already committed so there’s nothing I can do except build on top of them.”

In other words, the MVP software architecture plays a major role in allowing parallel development.

This situation in itself is not problematic because CM tools, most of the time, can handle most of the situations involving back merges. However, parallel development affects the testing activities that need to be repeated to guarantee that one’s changes do not interact in unexpected ways with the changes that had been checked-in before. As indicated by MVP developer-04:

“When I check-in … at that stage in the game I am looking at how many people are sitting here using it and also has anybody checked it in since I checked it out? ’Cause if that is the case, then I certainly need to get those changes and test again before I check back in. So if
three or four people have done work and checked a thing in since I checked the file out and now I am ready to check in and I have tested all of my changes, well I need to retest all my changes with the three or four different people who have checked in since then to make sure what they have done doesn’t change something that I’ve done.”

Another developer reported that he even tries to “speed-up” to finish his work sooner to avoid back merging and consequently new testing activities. According to him: “This is a race!”

To avoid back merges without avoiding parallel development, MVP developers perform “partial check-in’s.” In a partial check-in, a developer checks-in some of the files back in the main repository, even when he has not yet finished all the changes required for the bug fix or new functionality. This strategy reduces the number of back merges needed and minimizes the likelihood of conflicting changes during parallel development. In other words, MVP developers employ partial check-in’s to avoid being affected by other developer’s changes in the same files. The effect of these changes can lead to back merges or new testing activities. In short, a partial check-in is a form of backward impact management.

5.5.2. Handling External Dependencies in the MCW team

Backward impact management was particularly important to MCW developers because of the several dependencies that they had in different teams in the whole organization. I will call these “external dependencies,” to contrast with dependencies among the sub-teams of the MCW team. External dependencies are especially important, given the organization-wide reuse program adopted by BSC.
The first and perhaps more important strategy used by MCW developers was to use other teams’ APIs, which was in accordance with the BSC reference architecture and best practices in software engineering. By implementing against components’ interfaces, developers are minimizing the impact that changes in these components could have on their own work. Developers indeed took extra steps to guarantee that they would use other components’ APIs. For instance, a developer contacted another team to request them to extend their APIs in order to provide the services he required. Because of the different schedules being followed by BSC teams, sometimes MCW developers would even volunteer to make the changes in the code. This was especially useful when other teams were on a tight schedule for new releases of their products.

In addition to APIs, adapters were also used to wrap functionality, especially of other products in the organization:

“An adapter is an artifact that is deployed alongside a target component, and that manifests an interface that is useful to some consuming component. The adapter uses the target component APIs to do work. The consuming component communicates only with the adapter, and perceives a component that fits its desired interface.

Adapters are best used when there is limited or no control over the functioning of the target component; 3rd party of legacy components are typical candidates.”

Adopting APIs and adapters was a design decision taken by BSC software architects. However, software developers were the ones who actually talked to their colleagues and negotiated the exposition and/or implementation of new services in an API. In other words, although APIs were organizational practices to deal with dependencies, the communication, coordination, and negotiation to make them useful were actually performed by software developers.

In addition, I interviewed more than one developer who had switched teams to implement the services required by his own team. One developer, for instance, had
moved to another team that provided user interface components to the MCW team, with the unique goal of implementing the UI components needed by MCW client developers. He acted as the “contact person” and forwarded information about the new team that was useful for the MCW team.

In another occasion, a client developer “followed” his dependency in order to switch teams: he had a dependency on the server team, who actually had a dependency in the infra-structure team, who depended on an external team to provide the component. To simplify the communication channels and make sure that the client team would have the component when needed, the manager of the client team decided to “lend” this developer to this external team. This example also illustrates how the entire software architecture influenced the coordination of the work. It is not only the “direct” dependencies that affect a software developer’s work, but also “indirect” dependencies because those influence the direct dependencies.

Obviously, whenever possible, developers would contact their API providers to try to get information about the status of their work. Others would even attend these teams’ group meetings to learn about these teams’ plans and consequently assess how these plans would affect their own work. The information that these developers would get with their colleagues (whom they depended on) was shared with the other team members during MCW client or server team meetings. Another strategy adopted by the MCW team was to “group” requirements to make sure that all inputs were listened to and therefore the

---

25 This communication between API consumers and providers was possible when these developers were aware of each other. However, as I mentioned in the previous section, this was not always true.
design would not have to change later to reflect other needs. That happened with MCW developers who were providing and who were consuming new services. In general, MCW developers tried to anticipate changes in the code – that is, they tried to be aware of changes in the code they depended on, so that they could plan their own work accordingly. Again, this was only possible once MCW developers found out who was in their impact network.

5.6. Forward Impact Management

5.6.1. Problem Reports in the MVP Team

The organization of the work in the MVP team is centered around problem reports. To begin with, a MVP developer is usually delegated new tasks by being assigned to work with one or more PRs. This is usually done by email or through informal conversations between developers and process leaders or managers. These PRs had either been reported by the V&V staff or by other software developers. When software developers are reporting PRs, they might decide to divide a PR into multiple PRs that achieve the same goal. This division aims to facilitate the organization of the changes in the source code separating PRs that affect the released MVP tools from those PRs that affect tools or processes not yet released. By breaking a PR into several others, developers facilitate their colleagues’ work (especially that of V&V team members). When the V&V staff is reporting a PR, they initially search the database of problem reports to find out if the bug has already being reported. The convention used in the team is to report a bug according to the name of the function where the bug was found and that led the application to stop working. After that, if the bug has not been reported
before, they might talk to developers who work in the process in which the bug is likely to be found:

MVP-developer-03: “Usually, the process is that the PRs are filed through, you know, [bug-tracking] system but it is not uncommon before that PR has to be … PR has to be typed up and put into the system the person that finds it comes to see you. So if it is a [process1] bug [member1 of the V&V staff] who runs it operationally … would come and take this problem here with [process1] is doing this or someone else would mention it and I would go ‘OK’ and I would look at it and say ‘you’re right, it is doing that. It should not be doing that it should be doing something else,’ so she’ll say ‘shall I file a PR or is there a PR for this?’ I'll say, I will give her an answer but let’s say no, there is not a PR for that, ‘why don’t you file one?’ or I will file it and so either she or I will file it. In this case and if she files it, I will see through the email system that [member1 of the V&V staff] has filed a PR and then I will either do it myself or get some help.

Interviewer: “… people usually talk to you before filing the PR?”

MVP-developer-03: “Yes, normally because they want to know that it is not a duplicate PR. Normally they don’t want to file. If the same bug is reported three times, it is silly to file the same PR three times.”

Therefore, by contacting the “process” developers before reporting a PR, V&V team members minimized the work of MVP developers by avoiding duplicate PRs and also made them aware of new problems that appeared. Furthermore, the V&V team member who identified the problem is responsible for filling in the PR field “how to repeat.” This field describes the circumstances (data, tools, and their parameters) under which the bug appeared. With that information, developers can more easily perform their work.

According to the MVP process, developers should change the state of the PR to “work-in-progress” when they started working on a PR to indicate to their colleagues that work to implement the PR had begun. Despite the process prescription, MVP developers did not do that. Instead, they changed the state only when they were about to finish working on the PR. When this happened, an email was generated to all other developers making them aware that the developer was “about to check-in soon.”
To conclude the bug-fixing process, a developer fills in the other fields of the same PR, describing not only the changes he or she made in the code (through the designNar field, for example), but also the impact these changes are going to have on the V&V staff. The information about the impact on the V&V staff is recorded in the PR field “how-to-test-it” and in the one that describes whether the MVP manuals need updating. This information is used by the test manager, who creates test matrices that will be later used by the testers during the regression testing. The documentation expert also uses this information to find out if the manuals need to be updated based on the introduced changes the PR.

In some cases, developers will be even more specific:

“The developer will be very helpful and they will say Figure 7-23 in the [tool] manual needs to be changed. If they do that, it makes my job easier and I appreciate it, but I don’t expect it.”

Problem reports facilitate forward impact management among MVP developers and between these developers and the V&V staff. They provide information that is useful for different members of the MVP team according to the roles they are playing. They facilitate the management of impact (and dependencies) because they provide information to MVP team members that helps them in understanding how their work is going to be impacted.

5.6.2. Formal and Informal Code Reviews

Formal

The MVP software development process also prescribes the usage of code reviews. They are performed by “process” leaders whose “processes” are

---

26 Information about changes in the code are also used by process leads during code reviews, which are discussed in the following section.
affected by the changes in the code. Software developers need to request code reviews by email for accountability purposes. In addition, it is not unusual for developers to include additional information to facilitate the code reviews, for instance, by indicating that some files have been changed just to fix code standards and not for the main PR work.

Developers’ changes can be reviewed as many times as required until they are allowed to be checked-in. More specifically, according to the MVP development process:

“If the appropriate CSCI Lead(s) decide that the Developer’s code changes are not sufficient for the task, then the Lead(s) communicate with the Developer, then steps 6.1.13 through 6.1.16 are repeated until the CSCI Lead(s) decide that no further changes are required to accomplish the task.” [MVP Software Development Process Description]

Code reviews allow process leaders to be made aware of changes that might have “an impact downstream”: they can prepare for (or avoid) a possible impact that the changes being reviewed could have on these process developers’ work. Furthermore, as discussed by the MVP branch chief below, code reviews are also an opportunity to provide forward impact management because developers should redesign pieces of code that are getting too complex. This redesign is going to be beneficial to other developers in the future and perhaps to the developer himself:

“… in the last couple of weeks I have been writing my own code, and in the process found … was looking at quite a few files and saw some recent stuff that wasn’t real good … I see code that has been worked on over and over that has gotten so complicated because people are asked to do new stuff and they put it in and they don’t have time to rework what is already there, and really what should happen is we should redesign the whole piece so we don’t have a million IF statements.”
Informal

MVP developers will also contact process owners where they made changes to verify whether their changes are going to impact the work of these process owners. According to MVP Developer-04:

“when I am trying to test to see if it is really a bug sometimes you have to go say ‘how does this really work? [Developer’s name], what is the F9 panel for the [tool] panel do? What is it supposed to do?’ So that when I am fixing something I know how to test it now. So you go, ‘is that how it really how it is supposed to work. Is that really broke?’ … you got all these different versions of [process 1 - a process that deals with user interface] and you have people who specialize in it that work specifically with that particular mode of the [process1], so if I am working on something that looks like it might affect someone’s work then you might want to go down and get some input, ‘if I fix this this way, how does that affect your [tool] work?’… So you are asking the person that did the work and is specializing in it, ‘Is this really broke? Is that how it is supposed to work?’ and/or ‘I am fixing this, I am not breaking your work am I?’ … [a process leader’s name] is an example with the [tool]. If you are messing around in the [process1] and you are messing with [process2] and you start tweaking code that is going to affect him, it is probably a really good idea to touch base with him first and go, ‘Hey, I am messing around with this, it doesn’t look like it is going to affect you, is that true?’ If it is not, he will tell you. If it is, he will tell you. So you feel more comfortable with what you are checking-in, it is that one more extra step as you are doing your work to check with the people that are … because there are all these different groups doing their work and it is not always easy to keep up with who is who, but common courtesy again separate from ISO [9000] procedures and step-by-step, common courtesy says check with the guy doing the work that you are not breaking his work or you are not affecting his work. Sometimes you are in their way and they don’t particularly want to answer your question at the time, but as a rule at least you made the effort to say, ‘Hey, I am messing around in your sandbox here? My work has gotten me in your sandbox. Does that make sense to you? Should I be here? Or should I just pack up and take my changes and get out of here?’ Because there are times when people don’t want you messing: ‘Don’t mess with that there, that is exactly how I want that to work. If you do that, it ought to break. You shouldn’t be doing that.’ “

This quote illustrates how MVP developers try to minimize the impact that their work is going to have on their colleagues. By asking process leaders about their own changes, MVP developers make sure they do not impact their colleagues who “own” the process and are more capable of evaluating their changes.

Summary – Code Reviews

Formal and informal code reviews illustrate one important aspect of software development in the MVP team: its concern with maintaining the MVP software
architecture because of its complexity and specificity. Because of their understanding of the “process” they lead, process leaders can more adequately evaluate the impact of changes into the process’s architecture.

5.6.3. Holding onto Check-in’s

The software architecture of the MVP system again plays a role in the check-in process. For instance, as I found out in the interviews, some parts of the MVP software affect all other parts of the system so that changes in them will require most other developers to recompile their code, a time-consuming process. Knowing that, MVP developers often postpone their check-in’s in these parts until later in the day so that the other developers do not waste their time with recompilation. According to MVP Developer-04:

“People also know that if they are going to check-in a file they do it later in the afternoon. A lot of times you will find that people they do a lot of work, they are aware and most of us are here during the day and if you are going to do a check-in and it’s going to cause anybody who recompiles that day to watch their computer for forty five minutes, I mean most times you see that come in at two or three in the afternoon. You don’t see folks knocking those, I mean you don’t see someone doing a [change in a global file] check-in at eight in the morning because everybody all day is going to have to sit there and recompile … Most of the time you see that stuff in the afternoon so that it doesn’t … a common courtesy that you don’t have everybody else sitting around all day.”

This illustrates, again, how MVP developers engage in forward impact management: by postponing their check-in’s, they minimize the impact they inflict on their colleagues. This happens because these developers are aware that the size of the impact network is large.

5.6.4. Notifications in the MCW Team

MCW developers had an expectation that major changes in the code should be accompanied by notifications so that the build did not break. In fact, developers
reported notifying their colleagues in different occasions to warn them of major changes in the code and their associated implications. For instance, group meetings provided an opportunity to developers to inform their colleagues of changes that they would make that would impact the others. Developers also informed their colleagues on other teams. For instance, developers informed the installation team of new files being added or removed so that the installation procedures could be updated with this information. In other cases, server developers negotiated with client developers changes in one of the APIs that existed between the teams before actually performing the changes.\(^{27}\)

However, the usefulness of these notifications was contingent on knowing who to contact. As discussed in section 5.3, not all MCW developers were aware of their impact network; therefore, they were not able to provide important notifications. One strategy adopted by a developer was to read everything trying to find out what could impact him\(^{28}\):

*Interviewer:* “In your particular case, have you not received an email that you should have received? And because you did not receive it, have you wasted one day of work, for instance?”

*MCW Developer-15:* “Partly that. I sort of make up for that by reading everything. Obviously it is not a generically good solution because it means that you waste a lot of time. I basically stay in a hyper alert state constantly looking for things that impact me. The problem is that you read through a lot of things that you are not really interested in. I have reviewed a lot of these design documents [that I mentioned earlier] and I probably don’t ever have to necessarily need but I did not know if there was anything in there that was

---

\(^{27}\) Note that APIS are supposed to be stable to exactly reduce communication among developers.

\(^{28}\) Note that the quote below describes a backward management practice. However, this report is isolated among the data that I collected; that is, I did not find any other evidence of other developers doing the same thing. Because of that, I decided to classify the usage of notifications by the MCW team as a forward impact management strategy without a corresponding backward strategy. While it seems somewhat expected that other developers would adopt this same strategy, the lack of empirical data to ground this and the recommendations of grounded theory (Strauss and Corbin 1998) do not allow me to do so.
relevant to installation. ... Part of it is attention, being able to remind somebody that you are interested in what it is that they are doing. You are looking for a specific instance?"

Interviewer: “Yes. I am just wondering if you have one particular case.”

MCW Developer-15: “Let me give you an example. Our database developer [name] had certain files that were used to create databases. He changed the names of the files at one point so we lost some time while people were trying to deploy because they went to follow the instructions that I had written and they could not find the files that I was telling them to run. That is not the best example that I could come up with ... if I had ten minutes to search through all of my back emails, but I am not going to do that. But that is the flavor of the type of thing I am talking about.”

In a similar fashion, three different developers reported interest in being able to allow developers to inform colleagues of “critical updates” in the code that need to be picked up by all other developers using the code. For example, MCW Developer-05 said:

“If the developers can say ... this is the latest file, something that says that the other files that other people have are old ... and if this person only gets this one file and he may update and other people should get that latest one, if there is some way to mark that and it reflects on everyone, then you should get the new file which would be this one, I think that everyone should get this one. People wouldn’t be developing against this old file.”

Notifications are useful only to some extent: once an API is made public, control of who is using it is lost, and therefore notifications are not necessary anymore because these APIs cannot change anymore. Using the terminology that I proposed, the impact network became unknown. As described by a server developer (MCW Developer-10):

“We have latitude to change it [the API] as long as we are talking about an unpublished or semi-private API. If it is a contract between us and the client people, we probably have more latitude to change it and therefore they can trust it a little less than if it was a published API. At that point it would be very difficult to change it because people would be relying on ... right now we control everything that has a dependency, we control all the dependencies because the only people who are using the API are our own client teams and test teams and we can negotiate changes much easier than if they were external customers that were unknown to us or people in the outside world who we don’t control and who also could not readily change their code to accommodate our API changes. We would have to go about carefully deprecating, evolving, ah, end of life some features.”

On the other hand, developers are concerned about receiving too many notifications of things that were not relevant to them, especially when dealing
with discussion databases. That is, they were concerned about not being in one’s impact network and still receiving notifications of changes. According to MCW Developer-13:

“I think that in the beginning when it [the discussion database] was small, we used to go in everyday, at least I did, and look for new documents and keep updating. Now it is like, if someone has sent me an email that said that they have related a document and here is a link that is when I go to it. Because otherwise it is massive amounts of things and I cannot even make sense of it and how it is relevant to me.”

This occurred because of the size and number of discussion databases being adopted in the organization. In fact, this was a common problem reported by MCW developers.

5.7. Discussion

Impact management is the work performed by software developers to manage, control, and minimize the effect of software dependencies in their daily work. If two software development artifacts are independent, the developers dealing with them do not need to perform any impact management.

Given its definition, it’s not surprising to find out that the software architecture influences the impact management practices. For instance, MVP software developers recognized that the MVP software was not modular, and as a result, a change in one component could affect several others. Because of that, developers making a change in a module had to make sure that their changes would not affect other components in an unplanned way, and that changes in other components being done by the rest of the team would not affect their own changes. This was performed through error-checking activities, as described before. More important, components with a higher degree of coupling required more careful testing because of their larger “potential” for impacting other
components in the architecture. *This illustrates how the number of dependencies of a component influences software developers’ activities toward this component.* Furthermore, it also illustrates that software developers do know the degree of dependency in the MVP software. This knowledge comes with domain knowledge and/or experience with the project. In fact, MVP Developer-03, who mentioned that his testing is performed more carefully, has a master’s degree in a field closely related to the MVP.

In contrast, the MCW application had an architecture defined according to best practices in software engineering with controlled dependencies through layers, interfaces, etc. This architecture implied a small impact network compared to MVP developers. However, organizational factors played an important role in this case: the reuse program adopted by BSC made MCW highly dependent on components provided by teams who were located in other parts of the organization. On one occasion, another team in the company was not even aware that the MCW team required the component it was developing.

While API review meetings would allow MCW software developers dealing with these components to meet, changes in assignments disrupted this knowledge about the impact network. Furthermore, these meetings were useful only to the extent that API consumers and producers were invited to them. The distribution of APIs with “dummy implementations” allowed MCW developers to work independently. When integration was required, however, this independence became a problem because developers had different expectations about the API.

Another important aspect of the software architecture that influenced impact management was the extent to which an API was public. Before an API was
made available to other teams, it could be changed without problems. No one would be affected. When this API was “semi-public,” that is, available to only a small number of teams, changes had to be negotiated to minimize their impact. Finally, whenever the API was “published,” that is, made available to external customers, it could not be easily changed. Public APIs had to go through a slow process of changes in which some methods were slowly marked to indicate that API consumers should stop using them and start using the new API methods.

The MCW team also had to deal with problems caused by dependencies that affected them indirectly. For instance, the client team was dependent on the server team, who was dependent on external teams. If these external teams failed to deliver a component to the server team, the client team would also be affected. A strategy adopted was to lend developers to these external teams so that they could fulfill the MCW requests. In short, not only the “direct” dependencies affected the coordination, but also the “indirect” dependencies.

As illustrated by the ethnographic data presented, three other factors influenced impact management. First is the software development process. In the MVP team, this process prescribed important steps to be performed by developers to guarantee a smooth flow of work. Developers, however, deviated from this process to make it more adequate for their everyday work. In the MCW project, there was no formal software development process, but managers created particular approaches to handle their problems. For instance, the usage of API design reviews and the adoption of “pre-testing” activities simplified their work, providing both forward and backward management. Even though there was no detailed software process, managers, of course, had a schedule defining the number of iterations of development and integration activities.
necessary to deliver the software in the scheduled date. *In short, the accomplishment of impact management is embedded in and inseparable from organizational software development practices.*

The second aspect that influenced the impact management strategies was the software developers’ experience in the project. While the MVP project had been going on for more than nine years at the time of the study, the MCW project and the BSC organizational reuse program existed for little more than nine months! As one developer pointed out, this was not enough time to allow software developers to establish the social connections among themselves required for the accomplishment of their work. Similarly, MVP developers who had recently joined the project mentioned the importance of knowing who to contact when looking for bugs or changing unfamiliar “processes.” *This illustrates that the impact network changes while the work is being conducted.* Temporarily, the impact network consists of the “process” owner, who is the person more likely to be impacted by the developer’s changes. Once the changes are finished and are ready to be checked-in in the main repository, the impact network changes to the entire MVP team. This is also true in the MCW project, where changes in assignments lead to changes in the impact network. However, the frequency of these changes is very different in these two settings.

Finally, the last aspect that influenced impact management was the flexibility allowed by some of the collaborative and software development tools used. This will be discussed in the next section.
5.7.1. Tool Support for Impact Management Strategies

The tools used by MVP and MCW developers are described in Table 2 below with an indication of which strategy they supported. Note that some strategies are facilitated by several tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td>Learning from Email Notifications, Personal Network, Sending Email Notifications, Reading Email Notifications, Impact Descriptions, API Design Review Meetings (sending invitations), Formal Code Reviews, Notifications</td>
</tr>
<tr>
<td>Instant Messenger</td>
<td>Personal Network, Handling External Dependencies – Being aware by attending meetings, engaging in communication and so on.</td>
</tr>
<tr>
<td>Configuration Management</td>
<td>Error-Checking, Back Merges, Partial Check-in’s, Formal Code Reviews, Informal Code Reviews, Holding onto Check-in’s</td>
</tr>
<tr>
<td>Bug-Tracking</td>
<td>Problem Reports</td>
</tr>
<tr>
<td>Integrated Development Environment (IDE)</td>
<td>Error-Checking, The Reference Architecture and APIs, Pre-Testing Activities, Handling External Dependencies – APIs and Adaptors</td>
</tr>
<tr>
<td>Text Processor</td>
<td>Building Document, Handling External Dependencies – Group requirements</td>
</tr>
</tbody>
</table>

Table 2 – Tool Support for Impact Management Strategies

Software developers require a multitude of tools to support the different aspects of their work. To name just a few, IDEs to support programming activities, configuration management tools to coordinate access to the software repository, email and instant messenger tools to allow communication among team members, and so on. In this section, I describe how these tools facilitate or hinder
the accomplishment of impact management. A note of caution is necessary, though. As I mentioned earlier, impact management is embedded in and inseparable from organizational development practices. Therefore, it is not possible to analyze these tools without understanding the organizational context in which they are embedded. To illustrate this point, the software development process allowed MVP developers to use email messages for the identification of the impact network because these emails were broadcast to all software developers. The same thing did not happen in the BSC organization. MCW developers used email as well – indeed, a more modern email client. However, it was not recommended to broadcast emails to all developers involved in the reuse program. Nonetheless, developers still used email to notify their colleagues of important changes, and to contact friends who could potentially indicate who they should talk to about a certain component, among other things. Notifications were not particularly efficient due to the lack of knowledge about the impact network: those who needed to be informed were not, and those who needed to inform others did not know whom to inform. It is not that emails did not allow one to send the “right” information, but instead that one did not know to whom to send the information.

Overall, all the tools used by MCW and MVP developers were flexible enough to support the impact management practices adopted by these developers because they are indeed built to be flexible and usable in different settings. This suggests that it is more important to analyze the tools according to the analytical framework of impact management that I proposed (Dourish 2005). According to this framework, developers perform extra work to minimize impact from and into their colleagues’ work. However, as I have illustrated in the
previous sections, this is better achieved when these developers are aware of their impact network. The question that arises, then, is: how do these tools support impact network identification? I have already explained how email allows such identification. In addition to email, the CM and the bug-tracking tools can potentially allow this activity. However, again, organizational factors come into place interacting with the tool usage. Basically, at MVP, all developers used the same CM tool and shared the same code base. This CM tool was configured in such a way that all files contained information about all developers who had made changes in it. Despite that, MVP developers reported using email for network identification.

In contrast, in the BSC organization, the situation was different. Developers did not share the same code base because of practical limitations: synchronization with the main repository and compilation would take hours if that were the case. Even if developers had powerful enough machines to share the source-base, the CM tool would not completely support the identification of the impact network because of the usage of “dummy implementations” that were provided by the architects. In this case, if an API consumer would try to find out the author of the API (the one who is implementing the code that he was implementing against), he would find out the architect’s name, not the name of the developer who was actually implementing the API. In this case, the API consumer needed to be able to find which other developers depend on the same code that he depended on. This situation is hypothetical, but a real situation did occur: one MCW developer had to find another developer who was using the same component she was using.
A similar conclusion can be reached about the tools that support forward and backward impact management: they are flexible enough to support the practices, but they do not have (or leverage) knowledge about the software dependencies. As I illustrated, software developers use this knowledge to decide what to do next.

In particular, impact network identification was a major problem for MCW developers, leading to different coordination problems. Therefore, in the next chapter, I discuss how this task could be facilitated using Ariadne, a tool that I developed. This tool, of course, makes use of the dependencies among the software artifacts to support this activity. Consequently, it explores the relationship between software dependencies and coordination.

---

29 An exception is the CM tool, which has information about the software dependencies necessary only to decide, given a change, which files to build. CM tools are independent of programming language (Estublier 2001), and therefore cannot provide the type of dependency analysis used by software developers in their daily work.
Chapter 6 - Using Dependency Analysis with Ariadne

6.1. Introduction

The relationship between software dependencies and coordination has been studied by several authors (Conway 1968; Parnas 1972) and corroborated by both qualitative (Grinter 1998; de Souza, Redmiles et al. 2004) and quantitative evidence (Morelli, Eppinger et al. 1995; Sosa, Eppinger et al. 2002). In addition, the field studies described in the previous chapter highlight the practices used by software developers to minimize the effect of software dependencies in the coordination of their work. These results only confirm an unsurprising correlation: software engineers developing dependent pieces of code are more likely to engage in communication and coordination activities than developers working on independent pieces of code.\[^{30}\] What is surprising, however, is that such an obvious relationship has not yet been leveraged as much as possible to understand and facilitate software development, especially when such software dependencies may be automatically identified. Ariadne, the tool presented in this chapter, aims to support this kind of leverage. In other words, Ariadne aims to answer the second research question that I proposed, that of establishing the benefits of the usage of dependency analysis techniques in the coordination of software development efforts.

\[^{30}\] Note that, as other researchers (Morelli, Eppinger et al. 1995; Sanchez and Mahoney 1996) have pointed out, this relationship is not unique to software engineering.
Ariadne was designed based on the results of my field studies described in the previous chapter. It allows one to identify dependencies between software developers based on dependencies that exist in the source code. Ariadne’s underlying hypothesis is that, by revealing dependency information (between artifacts and between developers that arise from dependencies between software artifacts), it is possible to understand and facilitate the coordination of software development efforts. For instance, one can identify one’s impact network or one can infer which developers are more likely to interact with one another because of the dependencies in the code they produce (as suggested by Morelli et. al. (1995)).

The rest of this chapter is organized as follows. Initially, I will present the rationale for the design decisions that guided Ariadne’s development. Then, the following section presents an overview of Ariadne. After that, Ariadne’s main features are presented, followed by details of its approach. Finally, I present Ariadne’s architecture, describing its three main subsystems.

6.2. Design Decisions

Dependency relationships might exist between different types of software artifacts. As discussed in chapter 3, researchers have looked at dependencies between artifacts of the same type (such as software modules, components, and architectures) and between artifacts of different types (e.g., specifications and design documents). No matter the approach taken, the relationship between software dependencies and the coordination of the work still exists. For instance, dependencies between the MVP software and manuals led MVP developers to add fields to the PR that provided an indication of changes in the manuals. Similarly, the test plans created by a MCW developer during the pre-testing
activities were dependent on the integration scheduled negotiated between the client and server teams. If the scheduled changed, the test plans would need to be changed as well.

In this work, the type of software artifact I chose to support is source code because it allows the *automatic* identification of dependencies through the application of dependency analysis techniques. More specifically, I use the static call graph of Java applications.\(^{31}\) This allows me to identify all dependencies in the code and represent the strength of the dependency relationship between the Java modules; those are important aspects, as suggested by the analysis described in the previous chapter. Of course, it is possible to extend this approach by including dependency information among other types of artifacts. However, this information is difficult to obtain automatically (Spanoudakis and Zisman 2004).

In addition, I chose to extract information about software developers from configuration management repositories. Thus, I use authorship information to associate software modules with developers. Of course, this is not the only option available. In a particular context, one might want to use information about the last committer because he is considered an expert on the file (McDonald and Ackerman 1998). In a different company, where ownership architectures (Bowman and Holt 1999) are available, they could be used instead of authorship information because they express the relationship between developers and source code.

---

\(^{31}\) The static call graph is built using static analysis techniques, whereas *dynamic* call graphs are obtained by observing the actual execution of the program.
As I illustrated with data from my field studies, the impact network of a developer changes as his work progresses. For instance, in the MVP team in this network changed every time a developer started working on a process in which he was not an expert. Similarly, changes in work assignments in the MCW team led to changes in the impact network. One way of studying these changes is through the temporal analysis of these networks. That is, Ariadne supports the analysis of network given particular time stamps. In addition, Ariadne allows the user to select parts of the architecture to be investigated. The assumption, then, is that if a developer is required to make changes in a unknown subsystem, he can analyze only this system instead of being forced to analyze the entire project.

Finally, I chose to adopt a visualization approach for the presentation of the dependency information. Visualizations shift the load from the cognitive system to the perceptual system, capitalizing on the human visual system’s ability to recognize patterns and structures in the visual information (Robertson, Card et al. 1993). Currently, the visualization adopted is based on graphs, but other approaches are certainly possible. I also decided to present the entire network of dependencies among software developers instead of an ego-centered view. This way, indirect or second-order dependencies can be identified as well as opportunities for collaboration between developers with similar dependencies. Because of the large size of call graphs, even for medium projects, another

---

32 An ego-centric view only displays the ego-centered network (Wasserman and Faust 1994,p. 42).
33 This was inspired by a MCW developer in the U.S., who had to contact another developer in Japan only to find out that the developer implementing the component she needed was in the U.S. This developer was able to find this colleague only through an indication from the UI designer.
requirement was to be able to present these graphs at different levels of abstraction. Ariadne provides visualizations at three abstraction levels that are built by aggregating information from the inferior levels.

Before turning attention to Ariadne, I will very briefly describe a few social network concepts that are important for the understanding of this dissertation.

**6.3. A Brief Introduction to Social Networks**

A social network consists of a finite set or sets of social actors and the relation or relations defined on them (Wasserman and Faust 1994, p. 20). Examples of relations, also called ties, include family bonds, friendships, or shared workplace contexts (McDonald 2003), among others. It is also possible to represent information about different relations in the same network. In this dissertation, the relationship connecting software developers is one of source-code dependency. A developer is connected to other developers because his code is dependent upon those of these other developers.

Social networks are represented graphically by using sociograms. A sociogram is basically a graph containing nodes and edges. Nodes indicate social actors and edges indicate relationships between these actors. An undirected, or nondirectional, sociogram is one for which it is not possible to distinguish between the edge from $a$ to $b$ and the edge from $b$ to $a$. In a directed sociogram, this distinction is possible and is usually indicated with an arrow.

The concept of social network offers a way to analyze and understand the social interactions and organizations of our everyday experiences. Social network analysis (SNA), in particular, is an analytical tool that analyzes how patterns of
interactions among social actors form the structures by which they organize their actions and behaviors.

The main data in social network analysis are the relations among the social actors. Instead of focusing on the attributes of the social actors, the analytical focus is on the relations among them. For instance, among a group of developers, one could potentially record how easy or difficult is to coordinate the work between every pair of developers, or how often they engage in informal communication, and so on. In this case, a categorical scale could be used to represent the measurement of the relationship: very easy, easy, difficult, or very difficult.

An ego-centric network consists of the focal point (the ego), the set of “alters” who have connections to ego, and the measurement of the ties among the alters and the ego (Wasserman and Faust 1994, p. 42). An ego-centric network is a subset of the entire network.

6.4. Overview

Ariadne is implemented as a Java-based plug-in to the Eclipse IDE (Integrated Development Environment). It builds call graphs for any compiled Java project within Eclipse, and then it automatically connects to the configuration management repository associated with the Java project to retrieve its authorship.

---

34 Eclipse is an open source project that aims to develop a powerful IDE with an extensible architecture based on plug-ins. Therefore, one can create plug-ins to extend the IDE and still have access to all the resources provided by this IDE.
information. By combining dependency and authorship information, Ariadne generates a sociogram from the social call graph and displays it using by JUNG.\textsuperscript{35}

Ariadne targets two different types of users. First, project managers or researchers, who are interested in understanding the software development process. For example, by identifying bridges in the social network created by Ariadne, one can understand the key role played by some software developers in the coordination process or better understand developers’ coordination and communication needs. In fact, I used this approach to analyze open/free source software development (see next chapter).

Second, software developers, who are interested in identifying their impact networks and the set of developers that can impact this network. This information can be used for multiple purposes, such as to identify colleagues with whom they need to interact because of dependencies in their work, colleagues with similar dependencies with whom they could collaborate, developers that they depend on indirectly, and so on. In order to test whether Ariadne can be useful for software developers, I conducted qualitative interviews with commercial and open source developers, as also described in the next chapter. In this case, Ariadne uses the information available in a software developer’s private workspace. It takes into account the changes already made by the developer during the dependency analysis.

6.5. Main Features

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, it is integrated into this environment and makes use of several of the services it

\textsuperscript{35} Java Universal Network/Graph Framework, available at http://jung.sourceforge.net/
provides. For instance, it uses Eclipse’s SearchEngine class to extract software dependencies from a Java project’s source code. These dependencies basically represent a static call graph (see chapter 3). Figure 4 presents an example of a call graph for the open source project Tyrant built using Ariadne.

![Tyrant Call Graph](image)

**Figure 4 - Tyrant Call Graph**

Figure 5 presents a sociogram generated by Ariadne that describes dependency information among software developers of an open source project. This information is made available to any software developer using Ariadne, who can zoom in or out of the sociograms. The strength of the dependencies is indicated by the thickness of the edges between the nodes in Figure 5. In addition, one can select the strength of dependencies that one wants to display in the diagram. By doing that, we want to give developers the chance to choose the level of abstraction in which they want to look at the diagram.
Ariadne presents developers with three visualization options: technical dependencies, socio-technical dependencies, and social dependencies. Our current implementation can present technical and socio-technical dependency visualization at three different levels of abstraction, based on the programming language’s hierarchy (e.g., packages, classes, and methods in Java). Essentially, information from the dependencies between methods is aggregated to generate information about dependencies between classes, and, similarly, information about classes is aggregated to generate information about packages. Another reason for aggregating information is that different call-graph extractors might provide different results (Murphy, Notkin et al. 1998). By aggregating information, I expect to reduce the variability of the results provided by different call-graph extractors.

All visualizations provided by Ariadne can be exported as files formatted as comma-separated values. In addition, call graphs and sociograms can be exported to files suitable for use with social network tools such as UCINet.
Ariadne supports the temporal analysis of software dependencies in a manner similar to TeCFlow (Gloor 2004). That is, Ariadne can generate visualizations for projects of snapshots in time, which allowed me to study the co-evolution of a project’s technical and social dependencies. To demonstrate this feature, Table 3 presents graphs from three distinct releases of the open source project Megamek\(^\text{36}\). The graphs convey an idea of how Megamek’s architecture, presented as dependencies between its packages, is fairly stable between releases. In short, Ariadne can be used to study patterns of changes in software development projects. This analysis will be discussed in the following chapter as part of the evaluation of my approach. However, instead of focusing on call graphs, I will focus on the social dependencies among software developers.

\begin{table}[h]
\centering
\includegraphics[width=\textwidth]{table3}
\caption{Megamek’s Technical Evolution across Three Releases}
\end{table}

\(^{36}\) http://megamek.sourceforge.net
6.6. From Technical Dependencies to Social Dependencies

This section describes in more detail how Ariadne builds social networks by describing dependencies among software developers from call graphs.

According to Callahan and colleagues (Callahan, Carle et al. 1990), a call graph is a data structure that “summarizes the dynamic invocation relationships between procedures. The nodes of the call graph are the procedures in the program. An edge (p1, p2) exists if procedure p1 can call procedure p2 from some call site within p1. Hence, each edge may be thought of as representing some call site in the program.”

A call graph, then, reveals the latent dynamic structure of a software system, potentially unveiling dependencies among software developers responsible for these software components. For instance, assuming that a software component cA depends on another software component cB, and that cA is being developed by developer A and cB is being implemented by developer B, if cA depends on cB, we conclude that developer A depends on developer B. This translates into a need to populate the call graph with “social information.” The goal is to create a data structure that describes which software developers depend on which other software developers for a given piece of code (de Souza, Redmiles et al. 2004). An example of this data structure, called a social call graph, is presented in Figure 6. A directed edge from package pA to pB indicates a dependency from pA to pB. Directed edges between authors and packages indicate authorship information.
Figure 6 contains the reverse-engineering (reeng) subsystem of the Argouml\textsuperscript{37} project.

![Figure 6 - A Social Call Graph](http://argouml.tigris.org)

Because social call graphs describe both technical dependencies and authorship information, they can be used to generate social networks describing dependency relationships only among software developers. That is, they can be used to describe dependencies between software developers that exist because of

\textsuperscript{37}http://argouml.tigris.org
dependencies in the source code they are working on. Dependency information among software developers is presented by using a sociogram, a graphical representation of a social network. Figure 5, presented previously, is an example of a sociogram created using Ariadne.

Software developers can use these sociograms to find two important pieces of information: who they depend on, and who depends on their work – their impact network. As I discussed in the previous chapter, whenever a developer cannot identify his impact network, software development activities become more complex.

6.7. Architecture

6.7.1. Introduction

Ariadne was designed to be general enough to support various programming languages, configuration management (CM) systems, and visualizations. Ariadne’s architecture, in fact, allows multiple program dependency extractors, CM tools, and visualizations to be installed at the same time. It leverages Eclipse’s features to use the user’s context in Eclipse to determine which code generator and CM system is used to extract the relevant information required by Ariadne. This is achieved through the usage of a layered architecture presented in Figure 7. A layered architecture is organized in such a way that components in one layer can request services only from the components in the tier immediately below them (Rumbaugh, Blaha et al. 1991; Buschmann, Meunier et al. 1996).
The most important part of this architecture is the configuration management and dependency information API. This API is used to isolate the programming language and configuration management tools from the visualizations provided by Ariadne. Through this approach, independent developers can contribute new functionality (support to other configuration management tools and programming languages) to Ariadne while reusing previous visualizations. At the same time, it is possible to easily design new visualizations to already supported programming languages and CM tools.

To facilitate the understanding and usage of this API, Ariadne utilizes the façade design pattern (Gamma, Helm et al. 1995), which aggregates methods to query program dependency information, authorship information, and the combined information (the social call graph). For example, developers may query which classes depend on a particular class, who are the authors of a particular piece of code, how the ownership of a class changes from one release to the next, and so forth.
6.7.2. Subsystems

Ariadne is divided into three main subsystems: program dependency, configuration management, and visualization. In this section, I explain each one of them.

Program Dependency Subsystem

Ariadne has been designed to represent the relationship between programming language elements by using the composite design pattern (Gamma, Helm et al. 1995). This pattern allows the representation of part-whole hierarchies and a similar “treatment” of individual and composite objects. Being able to represent hierarchies in different programming languages facilitates making Ariadne independent of programming language. Ariadne’s API (Figure 8) uses the concepts of nodes and edges to model any graph and therefore any program in such a way that nodes can represent program elements (such as methods, attributes, classes, and packages in Java), and edges represent dependency relationships between these elements. The usage of the abstract class Edge allows one to abstract away the difference between the different edges in the visualization module, providing a generic way to draw edges. Furthermore, an edge can be queried for information about what piece of information it links. I describe the visualization subsystem in more detail below.
Using the program dependency subsystem, my colleagues and I (Trainer, Quirk et al. 2005) have implemented a code dependency infrastructure that analyzes Java source code and Eclipse's manifest and “plugin.xml” files. In this case, instead of using the call graphs, these files are parsed because each one of them contains its dependency information.

**Configuration Management Subsystem**

CM systems offer tremendous amounts of data that Ariadne aims to abstract into generic formats that developers can mine to produce visualizations. For the purposes of this dissertation, Ariadne models CM repositories in a generic way that allows views of a project’s data at one or many points in time, regardless of the CM system used. We believe we have designed an API that is generic enough to capture the essential functionality that Ariadne requires of systems such as CVS, Subversion, and Clear Case, while still providing detailed authorship information from repositories.
Ariadne’s configuration management subsystem is integrated with the dependency generator module so that it is possible to find dependency information for any versioned element. Ariadne uses this information to query the code dependency generator module for any language elements in the region.

Currently, Ariadne supports a CVS extractor that is used to automatically connect to a project’s CVS repository (using Eclipse’s Team API) and extract CVS annotations (changes and authorship information). My colleagues and I have also built an infrastructure that imports source-control annotations from Rational Clearcase™ (Trainer, Quirk et al. 2005). The results of the annotate command available in this tool are parsed directly into Ariadne and used to create the different visualizations that Ariadne provides.

**Visualization Subsystem**

Ariadne's visualization subsystem allows developers to explore dependency and authorship information. It provides visualizations of call graphs, social call graphs, and sociograms. This is done by using some of the same design principles found in JUNG, which makes it generic enough to display any type of visualization that can be represented as a graph.

Ariadne’s default visualization is a simple directed graph with nodes representing authors and edges representing dependencies between authors. Alternatively, the developer may implement the visualization of choice – that may be a line-oriented approach as in the SeeSoft project (Eick, Steffen et al. 1992), treemaps, design structure matrices (Browning 2001), or however else the developer chooses to visualize dependencies.

The following chapter describes how Ariadne was evaluated.
Chapter 7 - Navigating Mazes of Dependencies

7.1. Introduction

The goal of this chapter is to answer the second research question that I proposed, that is:

• What value can be derived from using dependency analysis techniques to help software developers to understand and perform their work?

This can be translated into the following hypothesis: By revealing dependency information (between artifacts and between developers that arise from dependencies between software artifacts), one can understand and facilitate the coordination of software development efforts. This hypothesis was tested in two different settings. First, I used dependency analysis to analyze different open source projects, aiming to understand their development processes.38 Second, I conducted qualitative interviews (McCracken 1988) with software developers to understand whether the information provided by dependency analysis could be useful to software developers to coordinate their work.

The aspects of understanding and facilitating software development work are described in the following sections.

38 This work was done in cooperation with other researchers, indeed other aspects of open source projects that do not include dependency analysis are described elsewhere (de Souza, Froehlich et al. 2005).
7.2. Understanding Software Development

The same principle used to build Ariadne was used to extend Augur (Froehlich and Dourish 2004), which was then used to analyze different open source projects. Overall, in this work my colleagues and I wanted to explore aspects of the relationship between software artifacts and software development activities as they are negotiated in distributed software development. This was done through the analysis of the content of software repositories.

7.2.1. Motivation

Although processes are more or less well defined in formal organizations, informal software development, such as that adopted in free and open source software projects, has a different character. Open source projects must essentially produce their own structures. These structures are emergent rather than formal, implicit in the development practice rather than explicitly codified (Scacchi 2002). Although this allows open-source projects to be flexible, it also makes them more complicated for participants to understand or explain. In our case, we wanted to find out whether dependency analysis could be used to understand open source processes. The rationale for this approach is based on the concept of inscriptions suggested by Latour and explained in the following section.

---

39 Note that even though the results presented in this section were not created using Ariadne, they were the direct result of the same insight that guides Ariadne’s design: dependency analysis of source code can be used to study and facilitate software development efforts.
7.2.2. Rationale

Drawing on anthropological studies of scientific laboratory practice (Latour 1979), Latour proposes the concept of “inscription” (Latour 1994) to describe how social arrangements, debates, divisions of labor, and patterns of work become inscribed into the artifacts and representations in which science trucks. Inscription is a process through which social practice and technological artifacts become inextricably intertwined. For example, standardized processes imply divisions of labor, standardizations of skill, and so on (Suchman 1983); formal models imply ways of uniformly disambiguating between “interesting” and “uninteresting” (or “relevant” and “irrelevant”) phenomena (Bowers 1992); and instruments and devices imply particular ways of working and available infrastructures (Star and Ruhkeder 1994).

Inscription issues are particularly relevant to software development because software artifacts are pure inscriptions; free from traditional physical constraints, they are written forms that describe the forms and patterns of software system structure and operation. Software mechanisms are, in general, subject to much less external constraint than physical mechanisms, which is a source of tension in joint hardware/software development teams. There are many different ways of producing working software systems. In short, software development is a particularly fruitful domain in which to study the relationship between technological artifacts and the social structures that shape them. It is this relationship that drives the work presented in this section. In particular, I present aspects of the relationship between software artifacts and software development processes.
Using a software visualization tool, Augur, my colleagues and I explored different open source software projects based on the hypothesis that the software developed by these projects would contain information about the process of development that led to it. Our results are described in the following sections.

7.2.3. Types of Projects

By looking at the sociograms of software developers as indicated by dependencies between code modules, we can see how different approaches to project organization are reflected, or “inscribed,” in the source code of each project (Akrich 1994; Latour 1994). In a centralized approach, the control is potentially reflected in a call-graph structure in which other developers’ code are called by a central developer’s code. This code is the “glue” that connects the whole project together, all other developers’ code do not interact among themselves. That is not to say that the other codes are not important nor relevant, just that the architect’s code is the one integrating the whole project. Figure 9 illustrates this example: it is possible to identify a high degree of centralization around developer “gt78,” the “architect” developer in this project. This figure is based on the analysis of the project ireport.40

40 http://ireport.sourceforge.net/
Figure 9 - Forms of Participation: Centralized.

Figure 10 illustrates a different structure, called densely networked, found in the project lucene\(^{41}\). In this example, instead of a single developer being responsible for integrating the whole project, now this responsibility is evenly divided among a group of six different developers with a high degree of interdependence between them. There is no central “architect,” but rather a group of developers interconnected. These densely networked projects are marked by a high degree of interdependence between different modules and developers, often approaching a “fully connected” state in which each developer depends on the code of all the others. The degree of participation may vary (it is rare for all members of the project to contribute equally, and a set of primary developers

\(^{41}\)http://lucene.apache.org/
normally emerges), but the developers cannot be easily distinguished in terms of their particular roles and responsibilities as developers.

![Diagram of forms of participation: Densely Networked](image)

Figure 10 Forms of Participation: Densely Networked

Finally, Figure 11 (project azureus\(^{42}\) shows a variation of the previous structures in which not only a core of seven strongly connected developers can be found, but also a medium-sized set of four other developers on the periphery of the project, that is, whose code does not interact. In this case, they are called the core and periphery divisions – a core phalanx of major developers is surrounded by a peripheral set of developers, less strongly connected. Note, again, that this is not a distinction between degrees of participation, but between forms of participation, as characterized by the dependencies of the source code. This is

\(^{42}\)http://azureus.sourceforge.net/
also not an arrangement in which a core group of developers is doing the majority of the work; rather, it is an arrangement in which a core set of developers generates code that is strongly interdependent, while a peripheral set of developers tends to be more isolated from each other.

![Image](image.png)

Figure 11 - Forms of Participation: Core-Periphery

### 7.2.4. Forms of Peripheral Participation

It is possible to further distinguish between various forms of peripheral participation. By tracing dependencies, one can see whether peripheral members are dependent on core members, or vice versa. Clearly, in some cases, the dependencies are mutual; these often characterize a peripheral developer who is playing a traditional role in the project, yet tends to be responsible for only some small portion of the system. More interesting, perhaps, are peripheral
participants whose connection to the core is a one-way dependency; either core modules depend on peripheral ones, or peripheral ones depend on core modules.

Dependency can be identified by calls from one component to another (or from the components of one developer to those of another.) So, peripheral modules that are called from core modules suggest a structure that is often associated with plug-ins, extensible component-based systems, or other systems in similar styles. In this case, a peripheral developer might develop a relatively self-contained module, which must be activated from the system core. I typically observed that the core developers, whose code is tightly interdependent, are associated with central functionality; the plug-in or self-contained module is peripheral in both functionality and in connectedness.

The inverse relationship characterizes a peripheral developer whose peripheral relationship is one of dependence on core functionality. Most commonly, we find this when a developer writes a test case, a novel user interface or application, or some other “wrapper” function that calls on or relies upon the functionality of the rest of the system. For instance, a Tyrant developer I interviewed reported that he makes changes only in small details of the program, not in its core functionality. This situation is described in section 7.3.

7.2.5. Core/Periphery Shifts

Earlier discussions of core and periphery focus on static structure, but in this case the interest is in the dynamics of the software processes, and in how participation shifts between core and peripheral participation. This phenomenon has been classified as both a learning and a political process, where one has to identify allies that back up a developer, “just like a statement in a scientific paper when it
is accompanied by a large number of references and citations” (Ducheneaut 2002).

This shift can be observed by examining the same open source project at two different moments. By looking at the dependency structures in the source code, we can identify a developer’s contributions and impact. In a shift from the periphery to the core, we expect to identify developers who initially contribute code that performs some function by calling others’ codes. As these developers become more and more important in the project, their code starts to be called by other developers.

Figure 12 and 13 illustrate this in the project Megamek. Initially, developer Hawkprime was located on the edge of the project, as measured by connections in the network. At left, he is connected to one other developer through his code (indicated by the directions of the dependency edges: from Hawkprime to the other). The reason for this is that BMazur is the principal interface author, and consequently more central than Hawkprime. Later (Figure 13), Hawkprime assumes a more central role in the project. Hawkprime is also a source of dependencies because he is the author of an interface being implemented by others; now other developers depend on his work. Furthermore, instead of being connected to only one other developer, he is now connected to six of them. Again, the shift can be noticed based on the relative importance of the code being contributed.

\[http://megamek.sourceforge.net\]
Figure 12 - “Hawkprime” in Periphery

Figure 13 - “Hawkprime” Shifts to Core
Using a similar approach, it was possible to identify the opposite effect, a developer’s shift from the core to the periphery of another project, ANT\textsuperscript{44}. This time, the developer Umagesh initially had the central participation in the project (Figure 14). This can be observed by the five edges directed to him in the graph. Later, Umagesh shifted to the periphery of the project (Figure 15).

As in previous examples, the important issue here is not so much that these shifts take place; the movement of people between peripheral and central positions is both common sense and empirically well observed (Ducheneaut 2002). The important issue is the way in which the shift can be found in the data; that is, that the pattern of participation is manifest within the inscription, and can be analyzed structurally through dependency analysis of the software artifacts.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure14.png}
\caption{“Umagesh” in Core}
\end{figure}

\textsuperscript{44}http://ant.apache.org/
7.2.6. Discussion

This work has been motivated by the question of whether aspects of an informal software process can be found in the structure of the software artifact itself. By analyzing software modules and software developers and their dependencies within the same frame of reference, this perspective highlights not just the links between people and code, but the links between people and others through code, and vice versa. This homogeneous analytic perspective is reminiscent of the actor-network approach (Callon 1986; Akrich 1994; Latour 1994). Actor-network theory maintains a deliberate agnosticism as to sources of agency, and insists that human and technological “actants” be given analytic parity. Latour (1994) points out that functions embedded in social settings may also be delegated to technology; for instance, “speed bumps” achieve the same effect as a policeman to monitor road traffic and ensure conformance with speed limits. Technological
arrangements, as much as social arrangements, can be used to produce control and conformance with social norms. Actor networks, then, bring together heterogeneous elements, including technologies, artifacts, and people.

One concept arising from this perspective on scientific processes is that of the “obligatory passage point” – a narrowing of the network that designates some particular element as one that must be navigated in order to achieve a result. As befits the homogeneous treatment of heterogeneous elements in actor-network theory, this might be any sort of entity. Professional certification might play such a role, for example; so might a particular theory, a scientific leader, a particular laboratory, and so forth. Using the approach proposed here, we can see how this theory can operate in open source domains.

In general, the relevance of these concepts to the work presented here is the light that they cast on the interplay between social and technical aspects in distributed activities. A number of authors have explored aspects of the social structure of open source projects (Mockus, Fielding et al. 2000; Ducheneaut 2002; Crowston and Howison 2005). Our approach has been to look at the ways in which aspects of the social and organizational structure are both inscribed into and achieved through the technological organization of the underlying artifact, the source code. The central lesson here is two-fold. First, although the rhetoric of open source is of openness and access, the practice of open source is about closedness and regulation; essentially, a central consideration in managing a successful technical project is to ensure the consistency and quality of the technological artifact under production, which is managed by vetting both contributors and contributions, and so a structure must be produced by which such a vetting can be achieved. Second, this structure is manifest collectively by the social and technical
organization of the project. The same engineering principles by which software systems can be organized to achieve technical properties (modularity, extensibility, robustness, etc.) are also ones by which activities can be partitioned and managed, and access to the system limited. What we see in these examples, then, is essentially the emergence of obligatory passage points within software development practices. Those points may be technical or human elements. As presented among the forms of peripheral participation, there may be a particular module or component into which others must be hooked; a dispatch table, an event loop, or some other.

That these structures should emerge in successful software projects is not surprising; these projects, after all, require careful coordination, and some mechanisms are needed to ensure that this takes place. That they should emerge within open source projects, while not surprising, is nonetheless interesting, in light of the open source movement’s focus on participation and accessibility. What is particularly interesting, though, is that these elements of software production can, themselves, be found within the software structures that are the focus of activity. Although Latour and others argue that processes and social structures are inscribed into scientific and technical artifacts, our experiences point to the ways in which, for software artifacts, they might be “read off” again. Our empirical examinations demonstrate that both software components and software developers can act, for example, as obligatory points of passage; the structure of a software project both reflects and constrains the development process.
7.3. Facilitating Software Development

7.3.1. Introduction

To find out whether Ariadne can be used by software developers to facilitate their software development activities, I conducted qualitative interviews with both open source and commercial software developers. In these interviews, developers were asked:

- First, to discuss relevant coordination problems they face because of technical dependencies and the strategies they adopt to handle these problems;
- Second, to discuss the results of the dependency analysis of their source code provided by Ariadne;
- Third, to comment on the visualization of software developers’ dependencies provided by Ariadne. The goal here is to find out whether Ariadne provided satisfactory information that allows software developers to find information that can help them in managing dependencies in their work; and
- Fourth, to comment on whether Ariadne would be able to reduce or provide an easy recognition of the coordination problems they need to deal with during their work.

Initially, I will present the evaluation performed with Tyrant open source software developers and then I will discuss the evaluation performed at MTD.\textsuperscript{45}

\textsuperscript{45} A pseudonym.
7.3.2. Tyrant

Tyrant\textsuperscript{46} is an open source project whose goal is to develop a “graphical rogue-like fantasy adventure game.” I interviewed Tyrant software developers by email after I analyzed their projects to produce call graphs, social call graphs, and sociograms. Tyrant’s sociogram is presented in Figure 16. It suggests a core/periphery structure by which developers in the core (chrisgri, cmuessig, jules_ve, mikera, and rickblai) are connected to other developers in the core and to the developers in the periphery (trump-ca, ppirrip, pmularie, jicksta, vicsot, tdemuyt, and itchykit). Developers in the periphery are connected only to developers in the core, but not amongst themselves.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{tyrant_sociogram.png}
\caption{Tyrant's Sociogram}
\end{figure}

We emailed this sociogram with a brief explanation of how it was built to Tyrant software developers, without discussing the idea of core/periphery with them.

\textsuperscript{46} http://tyrant.sourceforge.net
We asked them to comment on the diagram and whether its visualization would be useful during daily software development tasks.

One developer pointed out two interesting aspects. On the one hand, visualizing dependencies among software developers could help developers “shy away from certain classes,” which would actually be a good thing because some classes in Tyrant are used throughout the whole project and can potentially impact all other developers. On the other hand, the visualization of dependencies could be harmful because it would lead developers “to commit on stuff they shouldn’t commit on” in order to become more central developers and capitalize on that.

This same developer also mentioned that the five developers in the core of the network are in fact those that make more changes in Tyrant’s core. He reported that he tries to avoid touching Tyrant’s core and that he tries to make sure that his changes impact this core to a minimum and do not impact the periphery developers.

After his changes are finished but before committing them, this developer notifies the project lead and other Tyrant developers. I searched for examples of notifications in the developers’ mailing list and forum at Sourceforge, but could not find any examples of them. I tried to follow up this investigation with Tyrant developers to find out how these notifications are made and to whom they are made. However, for practical reasons, these developers needed to finish the interviews.
7.3.3. The MGR Study

The Organizational Setting

This evaluation was conducted in a very large software development company called MTD. MTD has sites spread around the world, and its clients include NASA, General Motors, Boeing, and similar companies. The project studied, called MGR (another pseudonym), is a relatively small project that started in March 2001 from the integration of three other projects within MTD. The MGR team is responsible for developing a web-based application that is integrated with two other large MTD products. Integration, in this case, is accomplished through standardized files exchanged among the three applications, not through source code.

The project staff includes about 18 collocated software engineers, the configuration management tool manager, the project manager, and six distributed software engineers (one in another part of the U.S., one in India, and four in Germany). Some of these developers are working not with Java, but with scripting languages; others do not work at all with implementation, but with installation, documentation, verification and validation, and other aspects of software production. In addition, sometimes these developers are pulled out of the project to work on new releases of the project that lead to MGR. In this organization, experienced developers are called architects. There are six architects in this team, four in Germany and two others in the U.S.

Weekly meetings are held with the collocated developers. Communication tools include Microsoft Netmeeting™, Lotus Notes™ for email announcements, and different instant messenger systems, for contacting the distributed
colleagues. Software developers use Eclipse, a state-of-the-art configuration management system and an in-house problem-report system.

Each developer located in the U.S. is assigned to three or four software components on a release-by-release basis, so that they can rotate functionality and get an overall understanding of the whole system. According to a developer, this makes the team less dependent on certain developers who can become experts in particular components and, for different reasons, leave the team.

The Software Architecture

The MGR software is being implemented according to the model-view-controller (MVC) design pattern (Gamma, Helm et al. 1995). This pattern organizes an interactive application into three separate parts: one for the application model with its data representation and business logic, one for views that provide data presentation and user input, and the third for a controller to dispatch requests and control flow. A fourth module is commonly used with this pattern, it contains information about persistence and data access. In the MGR team, this was not different. This pattern is implemented in such a way that the entire MGR application is divided into 152 different Java packages.

Methods

One of the developers interviewed used Ariadne to analyze the code of the MGR project. With that, it was possible to generate a call graph of the MGR code, as indicated in Figure 17. This figure was created by using pajek\(^7\) so that real package names were replaced by numbers and the structure of the code was

\(^7\)http://vlado.fmf.uni-lj.si/pub/networks/pajek/
reorganized to promote anonymity. Information about the strength of the dependencies between packages was also collected with the creation of the call graph, although it is not displayed in Figure 17.

Figure 17 - MGR Call Graph.

Configuration management information was extracted through embedded scripts available in the CM tool. This information was then parsed into Ariadne so that it could generate sociograms. However, one particular approach used by the MGR team influenced the results. In the MGR team, each individual developer is allowed to check-in code only in his or her private branch. After that, this developer is responsible for notifying the configuration management manager of this check-in. This manager is responsible for merging these results in an integration branch. Because of that, some developers call this manager the “gatekeeper.” Only after the integration is performed will the “gatekeeper” send an email to all other developers informing them that the changes are available. This email contains the "merge notes," that is, release information, the list of files
changed, the changes, and other pieces of information. After receiving this email, each software developer needs to synchronize his workspace with the integration branch, therefore incorporating others’ changes into his own environment. In this case, whenever a developer changes one file that is being currently changed by him, a back merge is necessary.

This approach for integration skews the results produced by Ariadne because when configuration management information about individual files is reported, it describes the “gatekeeper” as the main author of the changes in the code because he is the one who integrates developers’ code into the integration branch. The only exception to this situation occurs during release periods when developers are allowed to check-in code in the integration branch after receiving the authorization of the project manager. Despite the fact that the CM information I collected was skewed, it was still useful for my purposes because the sociogram generated by Ariadne is a graph almost fully connected, as indicated in Figure 18. Note that the resulting sociogram as presented below is extracted from NetDraw48 to replace developers’ real user id’s. In other words, the sociogram generated contains information about the developers’ work during release periods when they, instead of the “gatekeeper,” check-in code in the integration branch.

48 http://www.analytictech.com/netdraw.htm
In other words, the limited information collected about software developers’ check-in’s was already enough to indicate that the developers were almost fully interconnected: all developers were interdependent of all other developers. The only exception was an intern working in the team for about four months who only works with the presentation layer.

After using Ariadne to analyze the MGR source code and generate both call-graph visualizations and the respective sociogram, I conducted semi-structured interviews with four of the collocated MGR software developers. Questions in the interview followed the structure presented in the beginning of this section addressing coordination problems, the usefulness of the software dependency information, and then the usefulness of the information present on the sociogram produced by Ariadne. Questions also allowed participants to talk about their everyday work, including work processes, collaboration and coordination.
efforts, problems, tools, and so on. Interviews lasted between 45 and 75 minutes. Interviewees provided feedback on Ariadne’s results; this feedback is presented in the following section.

**MGR Developers’ Feedback**

Two developers reported that Ariadne potentially could be useful in “higher-up levels in the organization,” where teams would be represented instead of developers. According to Developer MGR-02:

“If instead of individual developers, those [in the graph] are groups then I see a big target for us.”

These developers reported that Ariadne could be useful in their organization because of the larger integration effort currently being discussed. This effort will include the MGR project and the other two projects related to it.

One developer argued that Ariadne could be useful for his own team, although it would “not make a difference” because his team is small and does not need to interact with other developers in the organization. In other words, identification of the impact network is not a problem for these developers. In fact, other practices adopted by this team substantiate that. For instance, during the weekly meetings, they discuss changes made in the code, so everybody is informed about what is going on. This happens even though during the interviews developers reported that they do not know exactly who their impact network is because developers change components’ assignments at every release. More important, a MGR developer reported that he does not need to know his impact network because, according to software process they adopt, architects evaluate major changes in the software and their impact in the software architecture. Minor changes, on the other hand, can be made by software
developers themselves on other people's code. This needs to be followed by a notification of these changes to the developers affected. If a change in another developer's code is necessary but is missed, this developer, after reading the email, will inform the author of the changes so that he can fix it.

Finally, when I asked another developer if he had an understanding of the system regarding which packages depended on which other packages, he told me that “he’s starting to learn more.” He works as the CM manager performing the merges from developers’ branches into the integration branch. This work helped him learn “which files to watch out for.” According to him, those files are in the core package of the MGR software, which is used by several other packages. When he notices changes in this core package, he expects problems with the developers’ merges. And these developers will come back to talk to him, because even if the merge was successful, there might be an unexpected behavior and “it is not functioning in the way the developers say.”

7.3.4. Discussion

Dependency Information and Coordination

In the interviews, I found out that the dependency information provided by Ariadne was helpful to software developers. The hypothesis that access to dependency information (between software artifacts and software developers) can facilitate coordination is, to some extent, confirmed by both Tyrant and MGR developers. As mentioned before, a Tyrant developer pointed out that the identification of classes that are sources of dependencies could be beneficial because it would indicate to software developers that changes in these classes
must be more carefully carried out. Similarly, the CM manager at MGR reported that he anticipates problems with the integrations whenever changes are made to the core MGR package. These results are similar to what I actually observed in the MVP team: processes with a higher degree of dependency were more carefully tested because they could impact all the other developers. Whereas in the MVP team, developers working with these processes are aware of their potential for impacting others, the Tyrant developer I interviewed suggested that Ariadne could be useful to unveil these areas in the software. The MGR developer reported another usage of this information: to anticipate and to facilitate his work as the responsible party for integrating other developers’ code.

Dependency information, in contrast, has to be examined in context. Open source software development is commonly regarded as a meritocracy: people ascend to positions of authority based on their accomplishments and expertise. In this case, making information about the *entire* social network of developers, as in my evaluation of the Tyrant project, might lead developers to work on the software modules that are the sources of dependencies: by checking-in code that should not be checked-in in these modules, a developer can be regarded as a core developer when he moves from the periphery to the core of the project. That, as pointed out by the Tyrant developer, might endanger the health of the project. In general, this confirms that one needs to be careful about the social information to be revealed using social network approaches (McDonald 2003; Gutwin, Penner et al. 2004). Note, however, that for researchers, as I discussed in the previous section, presenting the entire network is preferable so that one can observe structures and shifts between the periphery and the core. For the design of
Ariadne, this means that two possible visualizations are required, according to the type of user.49

Meanwhile, MGR developers pointed out that Ariadne is not likely to be useful for a small team like theirs, even though they change responsibilities every release. The small size of the team and the weekly team meetings allow software developers to be informed of the changes that can potentially affect them. This result is somewhat expected because one of Ariadne’s goals is to facilitate impact network identification. In small and collocated teams that adopt modular software architectures, such a problem does not seem likely to happen. However, according to these same developers, Ariadne could be useful for the large-scale integration project that is starting to take place at MTD. In this case, the large number of developers involved in this effort and the organizational and physical barriers for communication can potentially lead to problems in the process of impact network identification. To some extent, this context is similar to MCW’s context, where several developers were not aware of their impact network and because of that they either were not informed of changes that could potentially impact them or did not notify colleagues of changes they were making. Grinter (2003) reported similar problems in a similar organizational context, a large organization with different interdependent teams. All these results suggest that Ariadne, as a tool for software developers, is more likely to be adopted in similar organizational settings.

49 Another tool designed for researchers to study open source projects was proposed by Ducheneaut (2002).
Impact Management

Interviews with MGR developers provided additional insights on impact management strategies. First, a point raised by a MGR developer is that forward impact management can be performed only to a certain extent: a developer can not easily foresee the impact of his changes on another developer’s code if he is not an expert on this other component.

Another MGR developer reported that he does not even need to know his impact network because the architects in his team will evaluate his changes and he will even make the necessary changes in other developers’ code to fulfill his task. This suggests that the knowledge about the impact network is not always important; developers are indeed able to work without knowing their impact network. Most practices in the MVP team did not require developers to know exactly their impact network. These practices allowed developers to work without worrying about this: whoever needed the information was going to receive it because they adopted mechanisms to broadcast information among them. The only exception (code reviews) indeed occurs when new software developers face problems. On the other hand, in the MCW team, all practices required explicit knowledge about the network: developers needed to know exactly who to contact, even when interface specifications were used. Only temporarily, knowledge about the exact impact network was not required: in the beginning of the independent implementation allowed by these interfaces.

---

50 This point raises an important observation and led me to investigate my data one more time, according to what is suggested by grounded theory (Strauss and Corbin 1998)
In the MGR team, in contrast, the software architects might be performing forward impact management and potentially benefiting from tools such as Ariadne. Unfortunately, I could not interview architects to find that out.
Chapter 8 - Implications and Future Work

This chapter presents implications or recommendations that result from this dissertation. In addition, it describes some suggestions for further research.

8.1. Implications

8.1.1. Impact Management and Awareness

The term awareness has come to play a central role in CSCW, which led several researchers to explore how computational tools could support this phenomenon in different collaborative settings (Schmidt 2002). Unfortunately, however, “rather than motivate the investigation of this phenomena, this term has been used to gloss, rather than explicate, a range of activities” performed every day by competent actors while accomplishing their work (Heath, Svensson et al. 2002).

Impact management is the work performed by social actors to handle the coordination needs required in software development efforts that arise out of the (inter)dependent nature of software systems. Impact management strategies are then examples of the coordinative practices that the term “awareness” aims to describe. A parallel between these two terms can be traced:

1. Awareness involves two aspects: on the one hand, actors need to monitor the activities of their colleagues, and on the other hand, actors make their own activities publicly visible to others, including the monitoring of others. Furthermore, these aspects are complementary: “My monitoring the activities of others is facilitated by their displaying those aspects that are relevant for
me and my displaying aspects of my work to others presupposes that I am monitoring their activities and thereby am aware of their concerns, expectations, and intentions” (Schmidt 2002). Finally, awareness arises from the need to coordinate dependent tasks (ibid.).

2. Impact management also involves two aspects: on the one hand, software developers employ strategies to minimize the impact of their colleagues’ actions into their work, and on the other hand, software developers employ strategies to minimize the implications of their work into their colleagues’ work. These aspects are also complementary: whenever a developer tries to minimize the impact of his work into others’ work, he is also reducing the work done by others to minimize the impact of his work into their own work. Finally, impact management arises due to the nature of the work software developers are performing: their work is (inter) dependent.\(^{51}\)

In short, backward management practices illustrate how monitoring is performed by software developers. Similarly, forward impact management practices are examples of practices to display one’s activity.

In CSCW, awareness often is associated with particular types of subtle and supposed “effortless” practices – such as gesturing and gazing – that are required in settings like control rooms, newsrooms, and hospitals (Heath and Luff 1992; Heath and Luff 2000; Heath, Svensson et al. 2002). Impact management practices are, in contrast, definitely less subtle and require much more explicit effort from software developers: sending emails, running tools, ...\(^{51}\) In this dissertation, this dependency arises from the dependencies in the software artifacts being manipulated. However, it is perfectly possible to extend this concept to include the work done to minimize the impact of tasks where no dependency between artifacts exists.
attending meetings, etc. The distinction between these two categories of practices is based on the degree of explicitness. Schimdt makes this point more clearly:

because of the fine-graded repertoire of modalities of monitoring and displaying, ranging from something quite inconspicuous to something dramatically obtrusive, no clear distinction exists between, on the one hand, the coordinative practices of monitoring and displaying, normally referred to under the labels mutual awareness or peripheral awareness, and, on the other hand, the practices of directing attention or interfering for other purposes. In fact, by somehow displaying his or her actions, the actor is always, in some way and to some degree, intending some effect on the activities of colleagues. The distinction is not categorical but merely one of degrees and modes of obtrusiveness. (Schmidt 2002)

The distinction between the coordinative practices commonly associated with awareness and the ones that I have described arise out of the organizational settings in which these practices take place. Real-time settings require immediate action, whereas asynchronous work, as in software development, requires a different pace of activities. One should not expect a subway controller to send an email to his collocated colleague to inform him about problems in the schedule that need to be announced to the passengers; similarly, one should not expect a software developer to shout out loud his changes in the code that require all other developers spread throughout two different floors to recompile!

The implications of this dissertation lie exactly in this relationship between awareness and impact management. Because impact management practices are instances of coordinative practices by which software developers become aware of each other’s work, some of the same research questions worthy to be pursued
while studying awareness can be applied to the study of impact management, namely:

a. Upon which evidence does an actor rely when heeding the activities of others? What data (signals, cues) are available to the actor? What is the actor able to perceive of the actions of others? At which point does the actor seem to change course of action? Which specific situations or constellations of events seem to make actors change their course of action?

b. By virtue of which competencies are cooperating actors able to make sense of what others are doing? Which ‘taken-for-granted knowledge’ is invoked by the actor in making sense of the evidence available to him or her? Which ‘indicators’ or ‘typifications’ do actors primarily rely on? What do they monitor for and what is ignored? What is displayed and what is not? Which events make a difference and which are of no consequence?

c. How do actors exploit the material and conventional environment in monitoring unfolding events? Which indicators play a key role in determining the state of affairs? What is the relationship between the materiality of artifacts and their representational role as vehicles of signs? How is this duality exploited in monitoring and displaying?

d. How does the actor determine what is relevant to his or her own effort? How does the actor manage to sort out and pick up what is relevant? How does an actor, in modulating his or her activities so as to make relevant aspects thereof accessible to colleagues, determine what is relevant for the others? On the basis of which insight? How does an actor know when and how to ‘attune’ the ‘obtrusiveness’ of his or her monitoring or displaying? Based on such insights, we may be able to move the research on
computation environments to support awareness in cooperative work significantly forward. (Schmidt 2002)

In addition to these theoretical implications, the results of my dissertation suggest others. For instance, my discussion highlights how the impact network changes its size significantly during software development activities. More important, it suggests how features of the software architecture are influential in these changes in the impact network: the less modular the architecture is, the bigger the impact network. Therefore, it starts to suggest a relevant aspect that software developers use to determine what is relevant for their activities of monitoring and displaying.

Because a software architecture can be analyzed, another result of my work is the realization that dependency analysis techniques can play important roles in the identification of the impact network. In fact, it is theoretically possible to detect a developer’s current task at hand – which includes, but it is not limited to, the set of changes that a developer is implementing and the changes that other developers performed and that have not been incorporated into this developer’s workspace – to automate impact network identification. In other words, the results of this dissertation suggest that dependency analysis techniques could be used to suggest to a developer (i) the set of other developers whose actions he needs to monitor, and (ii) those developers who should be the recipients of his information display. This is particularly important in distributed settings, where the geographical distance does not afford the same opportunities for collaboration (Herbsleb, Mockus et al. 2000).

The relationship between dependencies and awareness had already been suggested by Schmidt (2000):
in depending on the activities of others, we are “not interested” in the enormous contingencies and infinitely faceted practices of colleagues unless they may impact our own work … An actor will thus routinely expect not to be exposed to the myriad detailed activities by means of which his or her colleagues deal with the contingencies they are facing in their effort to ensure that their individual contributions are seamlessly articulated with the other contributions.

That is, people have contextualized and different strategies about how they release their information, and they expect that others will do the same, not overloading them with information that is not relevant to their current context or activity. This suggests that, in software development, it is possible, again to certain extent, to justify why specific information is provided\(^{52}\) since the impact of one’s changes into others can be determined through impact analysis techniques (Arnold and Bohner 1993). Developers in the MCW, for example, questioned the relevance of receiving certain messages.

It is necessary to keep in mind two things, however. First, notifications sent to several developers might be important for other reasons. For instance, in the MVP team, it allowed new developers to learn about the expertise of the other team members. Second, implementation is only a small part of software developers’ work (Perry, Staudenmayer et al. 1994). Therefore, although approaches such as I suggested based on impact analysis might be extremely relevant in some settings, it is even more important to understand the underlying research question: How does the actor determine what is relevant to his or her own effort? I have started to provide a hint at the answer, using this knowledge

\(^{52}\) Palantir’s designers are exploring a similar approach, but for the purposes of parallel development (Sarma, Noroozi et al. 2003).
about the architecture of the source code. This in fact is an answer to another question asked by Schmidt: “Which ‘taken-for-granted knowledge’ is invoked by the actor in making sense of the evidence available to him or her?” The empirically grounded answer to this question is: his or her knowledge about the software dependencies that exist in the software architecture. However, more work needs to be done especially to understand software dependencies among different artifacts, where such knowledge is not so obvious.

The implications just described are the more fundamental ones. Others are possible as well, and I discuss them in the following sections.

8.1.2. On the Socialization of Software Immigrants

Dependency analysis techniques have long been applied to evaluate the quality of a software project, for instance, by calculating metrics that indicate how unstable a software component is and therefore to indicate if it needs to be refactored. As I described, experienced software developers also use this same information to orient their work toward a software module: more careful testing is required, the merging is potentially problematic, etc. With the course of the work in the project, software developers come to learn these components and the developers that work with them. However, new software developers are not aware of these intricacies (Sim and Holt 1998).

This suggests that dependency information about software modules and about developers can be especially helpful to software immigrants (Sim and Holt 1998) – new software developers who need to learn about the project’s history, terminology, work practices, code, and so on. This information can facilitate
learning about not only the code structure but also the associated social organization of the work.

8.1.3. Communication Barriers

Impact management illustrates the importance in software development of matching the software architecture and the organizational work practices (Conway 1968; Coplien and Harrison 2005). As illustrated by the MVP field study, a non-modular architecture was successfully maintained because communication barriers, for all practical purposes, did not exist among the software developers. In contrast, in the MVP team an architecture designed according to software engineering best practices had problems to be implemented because the software developers lacked social connections required to develop the architecture due to organizational and physical barriers. Those barriers complicated not only the communication among developers, but specifically the identification of the developers who should be communicating (Mockus and Herbsleb 2002).

Even best practices (such as interfaces), when used without being “supported” by organizational practices, can actually lead to problems. For instance, because “dummy implementations” of the APIs were provided along with the APIs, developers could work independently without communicating to each other. Changes in assignment of developers led to problems when API consumers and providers needed to communicate to integrate their work. Therefore, when communication was necessary during integration, it could not be established among those implementing the interface and those implementing against the interface. This result goes against the common belief in software
engineering that interfaces are enough to facilitate the coordination of the software development work:

Interface specifications play the well-known role of helping to coordinate the work between developers of different components. If the designers of two components agree on the interface, then design of the internals of each component can go forward relatively independently. Designers of component A need not know much about the design decisions made about component B, so long as both sides honor their well-specified commitments about how the two will hook together. (Grinter, Herbsleb et al. 1999) [emphasis added]

Others have suggested the need to allow communication channels among developers who provide and consume software artifacts in software reuse programs (Fafchamps 1994; Grinter 2001; Sosa, Eppinger et al. 2003). However, my results illustrate that this is necessary, or perhaps even more necessary, when interface specifications are used to avoid the misalignment of consumers’ and providers’ expectations (de Souza, Redmiles et al. 2004).

Finally, my empirical work suggests that unveiling dependency information – between software artifacts and between software developers – can also be beneficial to allow developers to find other colleagues with whom they could collaborate because they have similar dependencies. In other words, communication channels need to be established not only between developers providing and reusing components, but also between developers reusing the same components. This information needs to be carefully exposed, however (McDonald 2003).
8.2. Future Work

Impact management is the cooperative effort performed by software developers to properly manage the effect of software dependencies in the coordination of their daily work. I presented examples of these practices in this dissertation and illustrated how collaborative and software development tools hinder or support these practices. These results have expanded our understanding of collaborative software development, but there is still much work left to be done.

8.2.1. The Immediate Work: Extending Ariadne

Currently, Ariadne analyzes Java applications, which means that it is bound to use only during the implementation and maintenance phases of software development processes. Despite the fact that maintenance is the most expensive phase of software development (Ghezzi, Jazayeri et al. 2003), Ariadne’s support to the entire software development life cycle might bring the same advantages to other phases that it brings to implementation and maintenance. Therefore, an immediate extension to this work is an integration of Ariadne with software traceability tools, which formalize dependency relationships between artifacts generated during the whole software development process (Spanoudakis and Zisman 2004). Through this integration, impact network identification, as provided by Ariadne, could then be used throughout the whole development process to identify relationships between developers working with requirements, design specifications, and other documents.

Furthermore, Ariadne was designed with the main goal of allowing a developer to identify his impact network. However, as I discussed before, impact
management also includes two additional aspects: forward and backward impact management. Accordingly, Ariadne could be extended to support these aspects. Forward impact management could be supported through the analysis of current changes in a developer’s workspace to establish which other files and developers would be impacted. This information could be summarized in order to be sent to software developers. Meanwhile, backward impact management can be implemented by monitoring and analyzing files changed in the source-code repository. If these changes impact a developer’s code, Ariadne could potentially inform the user with notifications, decorators, and other types of visualizations. Integration among different instances of Ariadne could be achieved by using versatile event-notification servers (Silva Filho, de Souza et al. 2003; Silva Filho, de Souza et al. 2004).

8.2.2. Intermediate Work

Social Impact Analysis

Software engineers have been developing tools to facilitate the analysis of the impact of changes into other software artifacts. These approaches, broadly named change impact analysis, are divided in two main categories: program dependency analysis and software traceability (Bohner and Arnold 1996). They focus on “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change” (ibid; p. 3) with a technical perspective: the impact of the changes in the software application and in the other artifacts. For instance, a common approach is the identification of the set of test cases that no longer are applicable because of the changes in the software
source code (Rothermel and Harrold 1997). Using project management techniques, it is also possible to understand the effect of delays in the enactment of activities.

However, little research has explored the coordinative aspects of this analysis. For example, given the software architecture, it is possible to identify the set of components to be affected by the changes. If this information is associated with the respective developers associated with these components, it is possible to calculate the “coordination cost” of a change, that is, the number of software developers to be affected by a change.\(^5\) By overlaying information about the physical location of these developers, a manager could better assess the complexity involved in performing a change, especially knowing that changes involving distributed developers usually take longer (Herbsleb and Grinter 1999).

Exploring Multiple Social Networks

Several researchers have studied collaboration and coordination among open and free source software developers using social network analysis. For instance, Crowston and Howison (2005) examine the network that involves the task of bug-fixing: it contains core developers, co-developers who provide bug fixes, and active users who report bug fixes to understand. Lopez-Fernandez and colleagues (2004), in contrast, have looked at committer networks: each vertex

\(^5\) I am borrowing this term from MacCormack and colleagues (2004), who adopt a different definition, namely the measure of the degree to which information must be communicated between modules of related design elements, due to the dependencies that exist between these modules.
corresponds to a particular committer and two committers are linked when they have contributed to at least one common module. Finally, Gloor (2004) has looked at networks of developers created when these developers reply to each other’s emails.

Each one of these researchers has provided important insight into the phenomenon of collaborative software production in open/free software projects. Each one of these networks represents a different relation between software developers; it is possible to expect relationships between these relations. For instance, are alliances formed between developers during mailing list discussions “inscribed” in the software systems being developed? In social network terminology, is there a correlation between the communication networks studied by Gloor and the dependency networks generated by Ariadne? Are shifts from the core to the periphery in the dependency network preceded or accompanied by changes in the communication networks? Can one network be used to facilitate the understanding of and perhaps predict the other?

Note that this approach is not exclusive to open and free source projects. Commercial software development projects can also be studied and understood by exploring multiple social networks and the relationships among them. In fact, I have collected such social network data of a commercial software development team. This team, called MBL, is part of the same BSC organization to which the MCW team belongs. More specifically, I collected four different types of social networks. The first three social networks were ego-centric networks collected using questionnaires and latter aggregated, and the last network was extracted from the source code. These networks contain information about (i) the dependencies reported by software developers on one another; (ii) the awareness
network, to represent the extent to which a developer is aware of the status of his colleagues' work; (iii) the coordination difficulty between software developers; and finally, (iv) the dependencies between developers as generated by Ariadne, that is, dependencies between developers that exist because of dependencies that exist in the source code. Given that information, I plan to explore a number of different issues, including: Are the dependencies reported by software developers the same as the ones identified in the source code? Why? Why not? Does the awareness network match the dependency network reported by developers? Why? Why not?

Overall, what I am suggesting is the combination of multiple networks for the study of coordination in software development projects. In particular, dependency networks such as those provided by Ariadne can be automatically identified, reducing the problems associated with bootstrapping the network (McDonald 2003).

8.2.3. Long-Term Work: Back to the Research Agenda

Identification of New Strategies

As initially suggested in the research agenda presented in chapter 2, empirical studies of software development work should be conducted to identify software developers' strategies and hopefully to provide insights in how to extend and design collaborative and software development tools. In particular, the MCW project was in its first cycle of development, that is without any releases, while the MVP and the MGR projects were in the maintenance phase. Therefore, I believe that another study should be conducted with a team in the software
requirements and/or design phase. This is particularly important because these phases require, arguably, more collaboration among software developers (Herbsleb and Kuwana 1993; Olson and Teasley 1996).

The identification of new strategies is important, but again, specific for every team and organization studied. What is more important is that these studies can potentially shed new light on the concepts and aspects of impact management, especially how software developers manage their impact networks.

**Studying Co-Evolution: Software and Social Network Metrics**

Software metrics (Fenton and Pfleeger 1997), such as McCabe cyclomatic complexity, lack of cohesion in methods, and others, have been used to monitor or evaluate the quality of a software product. Similarly, social network measures (Wasserman and Faust 1994), such as density, centrality, and so on, have recently been explored by researchers to evaluate the health of teams, departments, and entire organizations: the extent to which organizational units rely on particular individual, how innovation spreads, and so forth. Some authors have also monitored software (Collberg, Kobourov et al. 2003) and social network (Gloor 2004) measures over time.

More important, however, is the potential relationship between code metrics and social network metrics. As discussed in chapter 3, several authors have recognized the relationship between software dependencies and task dependencies, and their effect on the coordination of cooperative work. By exploring the relationship between code and social network metrics, one can expect to understand the mutual alignment between software dependencies and coordination of tasks, that is, how these two elements co-evolve. For example, it
would be interesting to expand this research to study how structures of the code can provide insights pertaining to the development process to inform developers’ efforts and, conversely, how structures of an organization, and changes thereof, can offer information to help developers carry out their tasks.
Chapter 9 - Conclusions

9.1. Back to the Research Questions

In this dissertation, I have presented the concept of impact management, the work done by software developers to minimize both the impact of their work on their colleagues’ work and the impact of their colleagues’ work on their own work. Impact management is necessary to handle the effect of the software dependencies in the coordination of software developers’ work. This concept is grounded on the analysis of ethnographic data collected during two field studies that I conducted at large software development organizations. In this dissertation, this concept is described through the presentation of strategies that are interwoven into software developers’ daily work. Different from previous work (Staudenmayer 1997; Grinter 2003), impact management includes the everyday individual, ordinary practices that software developers employ to get their work accomplished.

The description of software developers’ impact management strategies aim to answer the first research question that I proposed in chapter 2:

- What are the individual strategies used by software developers to manage the effect of software dependencies in the coordination of their work?

An analysis of these strategies has also provided a discussion about the aspects that influence these strategies. More specifically, these aspects are the software architecture, the software developers’ experience within the project, and finally, the software development process and organizational practices of the team. In
general, the accomplishment of impact management is embedded in and inseparable from organizational software development practices.

The most important part of impact management is the identification of the impact network: the set of software developers that might impact one’s work and that might be impacted by one’s work. In fact, I have illustrated how the coordination of the software development work in the MCW team was problematic because software developers were not aware of their impact network. Impact management also includes two other complementary aspects: forward impact management and backward impact management – the work required to assess the impact of one’s change on the others and the work required to assess the impact of other’s work on one’s own work, respectively.

Because of the importance of the process of impact network identification, I developed a tool, Ariadne, to facilitate this process. Ariadne performs a dependency analysis of Java source code and extracts authorship information about this code from configuration management repositories. With this information, it builds a social network that describes a network of dependency relationships between software developers as indicated by dependencies between Java modules. Ariadne was created to explore the relationship between software dependencies and coordination. By unveiling dependency information, I expected to facilitate software development activities. With the results obtained, it is possible to answer the second research question:

• What value can be derived from using dependency analysis techniques to help software developers to understand and perform their work?
The results obtained while evaluating Ariadne answer this second research question. Information provided by dependency analysis of software systems can indeed provide insights about the coordination of software development work.

The reasons for this claim are twofold. First, through the analysis of software repositories, I exemplified how dependency analysis can be employed to understand open source software development activities. Basically, these activities become “inscribed” in the source code and can be extracted from software repositories through dependency analysis (Latour 1994). For example, it is possible to observe how developers move from the periphery to the code of a project and vice versa.

Second, through interviews with software developers, I learned how the information resulting from dependency analysis can facilitate the coordination of their work. For instance, this information can allow developers to be aware of certain areas in the code that might be particularly important because of their potential to affect the rest of the code. In combination with the results of the field studies, the results of the evaluation suggest that this information would be especially helpful to software immigrants (Sim and Holt 1998), new software developers who need to learn about the project’s history, terminology, work practices, code, and so on. Ariadne can then facilitate learning about not only the code structure but also the associated social organization of the work.

---

54 source-code + configuration management information.
9.2. Final Remarks

Developer-MCW-15: “So what you are seeing with other developers is that components have to mesh gears – just like the gears in a car engine, they have to be the right shape, size, and material and all of that in order to fit together. One developer has to know what another developer is doing so they know how to call the components that they depend on.”

As suggested by the quote above, dependencies between software artifacts play a fundamental role in the coordination of developers’ activities. The goal of this dissertation is precisely to investigate this relationship between dependencies and coordination by adopting an approach based on field studies and design of tools. The field studies allowed me to understand how software developers manage the effects of these dependencies in their work: to manage dependencies is to manage impact. Furthermore, this work is inherently interwoven with software developers’ activities, and therefore embedded in the work practices of every organization.

This understanding of impact management informed the design part of my dissertation. I used the insights from the field studies to build a tool, Ariadne, to support the investigation of coordination issues in software development. This tool was successfully evaluated and the results provided additional insights on software developers’ impact management work. Now, a new design cycle is necessary. In other words, Ariadne was only the first step toward the usage of software dependency analysis to understand and facilitate collaborative software development.
Chapter 10 - References


des Rivieres, J. (2004a) How to Use the Eclipse API.


Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley.


