UNIVERSITY OF CALIFORNIA

Irvine

Understanding Dependencies: A Study of the Coordination Challenges in Software Development

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Information and Computer Science

by

Rebecca Elizabeth Grinter

Committee in charge:

Professor Jonathan Grudin, Chair

Professor John L. King

Professor Rob Kling

1996

This dissertation of Rebecca Elizabeth Grinter is approved,

and is acceptable in quality and form for

publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine

1996

Dedication

To the British Taxpayer.

Thank you for giving me these opportunities.

Contents

Acknowledgments

I want to thank my parents for their support and their courage. They supported me by providing an environment in which anything was possible. When I chose to pursue higher education 5,000 miles from home they had the courage to let me go. My grandparents and my Auntie Dot have also encouraged me in all my endeavors. I love you.

My thesis is dedicated to the people who made this all possible, the British taxpayer. At the graduate level I was supported by the Engineering and Physical Sciences Research Council (EPSRC). I also thank the Fischer family and Hitachi for their support during my final year.

It was Jonathan who opened my eyes to the world of computer-supported cooperative work. His enthusiasm and insights into groupware and work made research come alive for me for the first time. This thesis hopefully reflects the passions I felt then and still feel for understanding the relationships between technology and work. Rob Kling introduced me to the world of social analysis when I arrived at UC Irvine. His insights gave me a new way of thinking about computer technology. John L. King sowed some of the seeds for this research in discussions about software project management. His own enthusiasm for the topic helped me to find questions that interested me.

The Ph.D. program is a collaborative endeavor and I owe the Turtle Rock group a huge debt for listening to my ideas, focusing my research, offering academic help, friendship, and support. JP Allen, Lisa Covi, Paul Forster, Roberta Lamb, Jeanne Pickering, and John Tillquist, I will never forget those evenings we spent working together. Nancy Eickelmann, André van der Hoek, David Hilbert, Holly Hildreth, David McDonald, Neno Medvidovic, and Leysia Palen also provided support and feedback on numerous occasions.

I owe a huge amount to Lisa Covi, Jim Whitehead, Marty Cagan, and Paul Dourish. Lisa's enthusiasm for research and insights into my work have helped shape this thesis and much more. Jim Whitehead not only introduced me to software configuration management but helped me find my research topic in that domain. However, it is his unfailing support of my work, even when I didn't believe in it, that I must really thank him for. I would also like to thank Marty Cagan, whose knowledge and insights into software development practice continue to shape and refine my ideas. Paul and I used the Internet to discuss our work, and through these conversations I rediscovered the purpose of this research. He helped me to find my confidence again and transform me from student to researcher.

My thesis would have not been possible without the cooperation of the three sites I describe, plus several others who did not appear. I would like to thank everyone for the time they spent explaining their software development work to me and their patience with my sometimes stupid questions. This thesis is a product of their collective wisdom.

Curriculum Vitae

REBECCA E. GRINTER
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717 USA
Internet: beki@ics.uci.edu
World Wide Web: http://www.ics.uci.edu/~beki/
Work phone: (714) 824-5086. Fax: (714) 824-4056.

EDUCATION

Ph.D.: University of California, Irvine
Information and Computer Science (1996)
Committee: Jonathan Grudin (Chair), John King, Rob Kling

Thesis Title: Understanding Dependencies: A Study of the Coordination Challenges in Software Development

M.S.: University of California, Irvine
Information and Computer Science (1994)

B.Sc.: University of Leeds, United Kingdom
Computer Science (1991)

WORK EXPERIENCE

1996 Research Assistant.

1995 Teaching Assistant, undergraduate Computers & Society course (including organizational and ethics issues).

SELECTED PUBLICATIONS

Grinter, R. E. (1995) "Using a Configuration Management Tool to Coordinate Software Development" *In Proceedings of the ACM Conference on Organizational Computing Systems (COOCS '95)*. San Jose, California: August 13-16. 168-177.

Blevins, J., Dubrow, D., Eickelmann, N., Grinter, R., Medvidovic, N., Reimer, R., Shaw, J., Turner, C. and G. Wong (1995) "Report on the Software Project Management Technical

Research Review" In D. Richardson and B. Boehm (Eds.). *Proceedings of the California Software Symposium*. Irvine, March 30, 1995. 127-145.

Grudin, J. and R. E. Grinter (1995) "Ethnography and Design - A Commentary" *CSCW: An International Journal,* 3: 55-59.

Pickering, J. M. and R. E. Grinter (1995) "Software Engineering and CSCW: A Common Research Ground" In Richard N. Taylor and Joelle Coutaz (Eds.) *Software Engineering and Human Computer Interaction: ICSE '94 Workshop on SE-HCI: Joint Research Issues* Lecture Notes in Computer Science Series Vol. 896. Springer-Verlag. 241-250.

Grinter, R. E. (1994) "Book Review: Review of and Perspective on "The Politics of Projects" Robert Block, *SIGOIS Bulletin* 14(3): 31-32.

Grinter, R. E. (1994) "Book Review: Review of "A Small Matter of Programming: Perspectives on End-User Computing" by Bonnie Nardi *SIGCHI Bulletin* 26(4): 80-81.

Grinter, R. E. and R. N. Taylor (1993) "Improvement of User Interface Development Methodologies through Rigorous Analysis" *UCI Technical Report 93-36*.

SPECIAL AWARDS, HONORS, AND OTHER INFORMATION

1996	CHI Doctoral Consortium participant.
1996	Fischer Fellowship recipient.
1992-1995	Engineering and Physical Research Council Studentship.
1994	Participant, Workshop on Software Process and CSCW, CSCW '94.
1991-1992	Selected as Education Abroad Program exchange student by the University of Leeds.

Abstract of the Dissertation

Understanding Dependencies: A Study of the Coordination Challenges in Software Development

by

Rebecca Elizabeth Grinter
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1996
Professor Jonathan Grudin, Chair

This research provides a new understanding of the dependencies that exist in software systems, and how software developers use practices and technologies to manage them. All software systems have dependencies because software modules interact with each other, with documentation, with libraries, and with test suites. Software engineers recognize that these dependencies exist, as technical relationships between the components of the system, and have tried to model them as part of their formal methods and process descriptions. However, no studies to date have examined the social aspects of these dependencies, how dependencies within the code, create and reflect social dependencies that exist between developers, teams of programmers, and software development organizations.

To address this issue I study the role of Software Configuration Management (SCM) practices and tools in the development process. SCM is the discipline of identifying the components of a software system and coordinating their development in order to control the evolution of the whole software system. Recently SCM practices have been embodied into tools that aim to support the development process itself. Using three interpretive studies I detail the different types of dependencies that exist during software development: why they arise, how they have both technical and social implications, and how developers and managers cope with them.

I use the findings from these studies to extend current understanding of how "groupware" technologies, like SCM systems, support the management of these software dependencies. I also highlight some of the problems in creating representations of dependencies, and consequently the times when SCM systems do not provide the required support to help developers coordinate their work. This understanding of how a technology supports the management of software dependencies contributes our knowledge about the role of systems in facilitating social processes, as well as opening up new questions about the extent to which that is possible.

Chapter 1

Introduction

> That's what delivering a software project feels like. All the cosmic tumblers have magically clicked into place cause you really don't know what's going to make it happen when you're doing it. You're just pushing back on every barrier removing any problem you know constraining it to something that's doable.
> Senior Project Manager, Computer Corp.

> All research starts from a *research problem*. ... The best sociological research, however, starts from problems which are also *puzzles*. A puzzle is not just a lack of information, but a *gap in our understanding*. A large part of the skill of producing worthwhile sociological research consists of correctly identifying puzzles. ... Puzzle solving research tries to contribute to our understanding of *why* events happen as they do, rather than simply accepting them at their face value. (Giddens, 1993; 676, italics in original)

1.1 Software Development and Software Failure

Double-click on an application, type in how much money you want at the ATM, or start your new car. Software running on computers, dedicated machines, or embedded into hardware, surrounds us. In the short time that has elapsed since researchers built the very first programmable machines software has transformed from specific scientific calculations written in arcane languages to generalized applications implemented in graphical development environments. Once a few select individuals wrote software for the machines that they used in their work. Today virtually all organizations buy or develop software to process information regardless of their business interests.

The increasing demand for software has created organizations that do nothing but develop software, such as Microsoft, Borland, and Netscape, and others that build hardware as well as systems, including Apple, Sun, and IBM. If we measure these companies' achievements by their ability to actually produce software, they are all successful. Despite these successes many software projects terminate abruptly, and sometimes in the glare of the media. These software projects fail at the cost of millions of dollars, with many jobs lost, and tragically sometimes with the loss of life.

In the 1990's alone several high-profile projects failed and part of the blame was assigned to the software: the baggage handling system at Denver International Airport destroyed bags, the Therac-25 machine killed people with high doses of radiation, the Airbus 360 fly-by-wire

airplane couldn't make a certain degree landing, London's automated ambulance transportation system left patients waiting for about half an hour, and both the California Department of Motor Vehicles and the London Stock Exchange spent money on software that never became operational. These are only some of the failures, those that were expensive or harmed people.

Why does software fail? This question led a group of researchers to a conference in 1968 where they founded a discipline, software engineering, that has worked hard to understand why software fails and invent better methods for developing systems. Like other researchers, I am motivated by the same question; however, instead of proposing a new software development technique this research focuses on looking at the practices of building systems. Describing how organizations build successful software provides insights into the challenges faced during development and strategies for managing those complexities.

Software fails to work for many reasons and this thesis describes just one problem that makes developing systems difficult. This thesis asserts that software development is difficult in part due the relationships that exist between software modules. These technical relationships create and reflect social relationships that exist between developers, managers, and organizations. The management of these relationships is critical to producing systems, and when they are ignored or misunderstood, the chances of producing working software decline.

1.2 Taurus, a Failure to Manage Dependencies?

The Taurus system was sold as a multi-million pound project that would revolutionize the London Stock Exchange. Since the unification of European Community, London — a traditional financial center — has found itself increasingly competing with other cities for the high finance business. Today the financial district finds itself competing with more modern exchanges: organizations that use technology to help them make swift transactions.

The London Stock Exchange decided to modernize its operations by introducing new technologies both within the exchange itself and among the organizations associated with it (Green-Armytage, 1993). The project started with all the optimism of any new venture sold as revolutionizing an industry. However, in 1993, after years spent in development the head of the London Stock Exchange announced that the Taurus system had failed. It never reached the operational stage; at its demise parts of it had not been implemented.

In the initial days after the announcement people searched for answers: why had Taurus failed? A highly regarded firm of computer consultants, Ovum Consultancy, suggested that the failure of Taurus was a direct result of poor configuration management practices. Configuration management involves identifying the components of a software system and tracking the changes made to them. It also involves maintaining information about how to assemble the components into systems. In practice developers and organizations find configuration management activities very difficult because the software components have technical relationships — called dependencies — that must be coordinated by the people working on that code.

Taurus was a highly distributed system; teams of developers worked together on individual parts of the projects. At the same time Taurus required that all the organizations linked to the London Stock Exchange build Taurus compliant systems. In this highly distributed environment the different developers and organizations struggled to coordinate their efforts with each other. Technically, it was difficult to align the distributed development efforts so that all the systems worked together. Socially, it was hard to maintain communications among the different developers and organizations working on the project so that everyone understood what changes were taking place and why.

Software engineering researchers know that relationships exist between pieces of code; they call them dependencies. However, little is known about how technical dependencies among modules of code create and reflect social dependencies among the developers, teams, and organizations working on them. The story of Taurus clearly illustrates that the problem of trying to coordinate these technical dependencies is a managerial problem. This thesis begins with a puzzle, the puzzle of understanding how technical dependencies in software create and reflect social relationships among developers, groups, and organizations. This research explains how successful development organizations manage both the technical and social aspects of these dependencies in the production of software systems.

1.3 Research Question

My research question is:

How do software dependencies affect the development of systems?

The research question is divided into three parts:

- What are software dependencies?
- Why do they occur?
- How do developers and organizations cope with these dependencies?

To answer the question of how software dependencies affect the development of systems requires that I explain what software dependencies are (relationships among code, people, and organizations that have technical and social aspects), when they occur (as a result of external influences on the software development process, or because software modules depend on one another) and how developers and organizations cope with them (through the implementation of configuration management practices and tools).

Software engineering researchers have recognized that dependencies exist, but have focused on their technical aspects. The software project management literature describes the role of coordination during development, but has not really asked why developers need to coordinate their work. Researchers interested in software project management have described a variety of strategies that suggest that developers do coordinate, that people need to know what others are doing, but few studies have looked beyond these observations to understand why this

collaboration is necessary. This thesis provides an explanation of one of the reasons why developers have to coordinate with each other: to manage dependencies.

Software configuration management concerns itself with the identification and control of individual components, their relationships with each other, and the change of the system during its evolution. While configuration management experts have proposed a variety of normative procedures for doing it in practice, few researchers have answered the question of why configuration management is important critically. I claim that configuration management — as it has been constructed in the textbooks of normative procedures and goals — is the discipline of managing the *technical* aspect of dependencies.

The data presented suggest that configuration management in practice involves the on-going management of a myriad of *technical dependencies* that create and reflect *social dependencies* between individuals, groups, and organizations. A simple framework for understanding the different types of dependencies that occur, among individuals, groups, and organizations is described. It distinguishes these three types of dependencies as the scale of their reach across the organization varies. Finally evidence is introduced suggesting that the size and type of software development process in place influences how these dependencies manifest themselves and the coping strategies required to manage them.

1.4 Software Dependencies

Software engineers know that dependencies exist between modules; after all, these relationships are a consequence of modular design. Modular design has had a profound impact on software engineering, and to understand how software engineers understand dependencies, we must begin with an examination of this practice. Modular software development involves breaking down a problem into its logical components and constructing a solution for each part.

Although modular design existed in practice, one of the first people to discuss it formally was David Parnas. In 1972 Parnas wrote an influential paper describing the criteria developers should use to divide a system into modules. He provided the foundation for a stream of research exploring different ways of deriving modular systems from the overall specification of the software. The fact that software engineers now call units of software code "modules" reflects the importance of the idea that systems should and can be broken down into tractable units.

For software engineering researchers a good modular system has certain features, including low coupling of modules. Coupling,

> measures the interdependence of two modules (e.g., module A calls a routine provided by module B or accesses a variable declared by module B). If two modules depend on each other heavily, they have high coupling. Ideally we would like modules in a system to exhibit low coupling, because if two modules are highly coupled, it will be difficult to analyze, understand, modify, test or reuse them separately. (Ghezzi, Jazayeri, and Mandrioli, 1991; 51)

This definition of coupling reveals several things. First, software engineers clearly recognize that dependencies exist. Second, they view them in a purely technical way; for example, dependencies exist when variables get passed between two modules, or one module calls another. Third, the difficulties of having dependencies have purely technical impacts, these relationships interfere with making changes, software reuse, or testing. Fourth, an appropriate solution for managing dependencies involves designing the system with as few of them as possible.

These researchers have identified a critical part of dependency management, the purely technical issues, but their account is unsatisfactory for three reasons. First, they have not identified all the sources of dependencies. Some dependencies come from outside the organizations, because code built by an organization relies on code built by other vendors, for example. Such dependencies may have technical impacts on the software development process, but cannot be resolved by designing for low coupling. Second, leading from that point, their solution does not consider the fact that the design of software changes throughout development. In practice developers find it extremely hard to design the final product in the initial stages of development. As modules are extended and adapted during the development process, initially low coupling may change over time. This problem is much worse when systems development begins with existing software — legacy code — and the developers must extend and modify its functionality. Finally, and most critically, it takes no account of all the social processes at work that conspire to make software dependencies even more complex, which the rest of this thesis argues is critical to building working software.

1.5 Summary of Thesis

Chapter 2 describes the directions of software engineering research. It begins with a description of the first conference held to establish software engineering as research discipline. It focuses on the evolution of software project management research, and discusses what is known about the coordination required to manage the development of software. Chapter 3 describes the history of configuration management as a practice for controlling the evolution of hardware and later software. It also discusses the emergence of configuration management systems; technological support to help maintain control over the development of software. These systems share features with other kinds of groupware technologies that are also discussed in this chapter. It concludes with a discussion of the data gathering and analysis methods.

Chapters 4, 5 and 6, introduce the three sites in the study: Tool Corp., Computer Corp., and Contract Corp. Each chapter describes how that organization (at the level of the individual, group and organization as a whole) coped with the dependencies that arose in their configuration management work. It shows how the social aspects of dependencies make configuration management extremely hard to do in practice. Chapter 7 synthesizes the observations presented in Chapters 4, 5, and 6. It describes the sources of these dependencies. Dependencies arise because: systems evolve over time, external influences force software to change, there is a continual need to reassemble the whole from the parts, and finally because organizations have to

build multiple products at the same time. Chapter 8 discusses future work, the limitations of this research, and concludes.

Chapter 2

The Software Crisis Becomes a Software Depression

> The fact that the crisis is still with us, over 20 years later, should tell us two things. First, the software production process is not like traditional engineering. Second, the software crisis should rather be termed the software depression, in view of its long duration and poor prognosis. (Schach, 1990; 5)

This chapter sets the background for this thesis work by examining the challenges of software development. It begins by introducing the motivation for establishing a discipline of software engineering, and introduces the problems that researchers thought needed solving in requirements, design and development, measurement, testing and project management. The chapter focuses on one aspect of software project management: the coordination of software developers. Although software project management research has identified the importance of coordinating the development, it has not substantially answered why developers need to work with each other. This chapter reviews research that suggests that dependency management is one reason why coordination takes place and reviews observations about dependency management in practice.

2.1 Software Engineering and the Software Crisis

In the autumn of 1967 the NATO Science Committee met to discuss computer science. During the discussions they became particularly concerned about problems that NATO members faced developing software. The Science Committee established a Study Group to assess these problems and make recommendations about how to continue. In late 1967 the Study Group recommended that there be a working conference to discuss the challenges of software development.

On October 7-11, 1968, researchers met in Garmisch, Germany, to discuss software development (NATO, 1969; 13). The participants were carefully selected, from academia and industry, for their knowledge and understanding of the software development process. Their brief was to discuss the problems of building software and establish a research discipline that they called "software engineering."[1] As the report explains,

> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of

[1] Although the conference was sponsored by NATO the discussions reveal that the mix of industrial and academic attendees kept the focus on software broadly, including commercial development, rather than just military concerns.

> theoretical foundations and practical disciplines, that are traditional in the established branches of engineering. (NATO, 1969; 13)

Software development began long before the Garmisch conference was convened. Most of the initial systems were military command and control systems; however, companies including IBM had also begun developing operating system software. During the 60's, companies involved in developing military and commercial systems started to discover that building software was difficult, and that as systems got larger and contained more interactions, the development process got more complex. It was these difficulties that attracted the attention of the NATO Science Committee, and led them to organize the conference.

At Garmisch the participants referred to the challenges of developing software as a "software crisis." Software engineering and the software crisis have remained tightly coupled since the conference; as the challenges of developing software have persisted so has the idea of the software crisis. The conference itself has become important to the software engineering community marking the formation of the discipline.

2.2 Initial Explanations of the Software Crisis and Hard Problems

The participants gave four reasons why the crisis had emerged: a lack of experience in developing software, economic pressures to build complex systems, the inherent difficulties in software production and problems monitoring the development process itself. Although all of the attendees at the conference had developed software, some of them expressed concerns about their lack of experience in building systems. This lack of experience was exacerbated by the limited opportunities they had to discuss their difficulties with other software developers and project managers.

Some individuals pointed to the economic drivers behind software development as potential causes of the crisis. Organizations had started to want computer-based solutions for their problems; for example, as the amount of flights in Europe increased the aviation authorities began to investigate the possibilities of automated air-traffic control systems. At the same time the functional complexity of the applications that organizations required increased. The participants argued that these demands for software were forcing development organizations into situations that were beyond their current understandings and abilities.

The participants also felt that the production of software required high levels of research and innovation. Hardware and software changed so rapidly that even upgrading existing applications often meant building a new product. Instead of being able to build on previous experience, software developers found themselves having to reinvent the system on the new and less well-understood platforms.

Finally, the participants recognized that they could not easily assess the state of development. No-one could accurately predict how long a system would take to build, how complex the

software was, and what size it would be when completed. It was hard to tell where in the development life cycle they were at any given time.

Having discussed the software crisis and the reasons why it emerged, the participants focused on key problem areas that needed addressing. I have named these areas the "hard problems" of software engineering because the participants clearly felt that if they could solve them then the software crisis would dissipate. The hard problems form the backbone of the modern discipline of software engineering and include requirements, design and development, measurement, testing, and project management.

2.3 The "Hard Problems" of Software Engineering

All students learn about the hard problems of software engineering in classes and textbooks (e.g., Sommerville, 1989; Schach 1990). The details of the problems and their partial solutions have evolved as the technology and techniques available have matured. For example in the 1990's students learn about the difficulties in designing client/server technologies while back in the 60's they would have focused on mainframes. However, the character of the problems has remained the same since the conference. The hard problems still form the guiding principles for much of the research within software engineering.

Requirements

The conference participants discussed the difficulties of eliciting requirements from users and customers of the system. At the same time they recognized the need to involve these groups, and felt that current software development remained too isolated from the environment that the system was expected to work in. Finally, they also observed that requirements change during the development of the system itself. Requirements analysis, definition, and elicitation have become important research topics since the conference.

Design and Development

Concerns about design and development permeated the conference.[2] They focused on defining and arranging the steps in the development process. Various aspects of development were discussed in isolation including the following: the merits of top-down versus bottom-up design, notion schemes for describing the system structures and states, and criteria for design like flexibility, design for change, usability, reliability and completeness.

---

[2] Researchers use these terms inconsistently. In my discussion I refer to development as the entire process of building software, from the initial inception to the final product. Design forms one aspect of development, although in practice it gets repeated throughout the entire development life cycle. Design focuses on mapping out the product, conceptually -- what it does -- whereas implementation concerns itself with how the product actually achieves the goals of the design.

Design and development issues remain at the center of software engineering research. Since Garmisch the attentions have shifted and extended as software researchers have discovered new ideas, and appropriated technologies in pursuit of resolving these goals. High-level languages, Parnas's (1972) work on modularity, object-oriented design (Gamma and others, 1994), parallelism, prototyping, Boehm's (1988) spiral model, are a few of the ideas that have contributed and extended this research.

Measurement

The inability of managers to measure progress during software development was discussed as a contributing factor in the software crisis. Subsequently, software measurement has become another stream of research in the software engineering community. Researchers have developed complex schemes for cost estimation (Boehm, 1981) and software complexity (McCabe, 1976). More recently researchers including Basili and Musa (1991) and Potts (1993) have called for the development of experience laboratories, where researchers can garner metrics from real-world software projects.

Testing

By the time of the conference, complete system testing required more resources than most organizations had to spend on the activity. All the attendees were concerned with testing the performance, reliability and accuracy of the software. At the same time they also discussed the importance of testing the system with its associated hardware and documentation. Testing has established itself as a critical part of software engineering research. Research has focused on defining subsets of the software that when tested capture all cases and states that the system can get into and explored the role that technological support can play in comprehensive system testing.

2.4 The Hard Problem of Project Management

The state-of-the-art in requirements, design and development, measurement, and testing has advanced since Garmisch. Although they all present important and exciting research challenges these topics are not addressed directly in this thesis. Instead this work builds on the hard problem of software project management. In this section I review the challenges of software project management.

Project Management at Garmisch

The Garmisch participants had plenty to say about project management. They observed that there was a high degree of variation in ability between individual programmers. However, they did not describe the skills that good developers had.

Most of their project management discussions focused on the difficulties of managing large software development efforts.

> More than twenty programmers working on a project is usually disastrous. ... We must learn how to build software systems with hundreds, possibly thousands of people. ... it is quite clear that when one deals with a system beyond a certain level of complexity, e.g. IBM's TSS/360, ... the sequence of changes that one wishes to make on it can be implemented in any reasonable way only by a large body of people. (NATO, 1969; 68)

Specifically they asked "how can we organize people so that they produce software efficiently?" Their solutions focused on improving communications among project participants. They believed that good communication was vital to establish and maintain control over the development process itself.

One participant suggested hiring friends to work on the same project to guarantee good communications among developers. Another proposed using deep and narrow organizational hierarchies for coordination; no individual should have more than five direct reports. Project meetings and code reviews were also suggested as ways of ensuring that large groups maintained vital communications while developing software.

Although the strategies for coping with the difficulties of developing software in large groups may seem rather simplistic and naïve, the participants of Garmisch identified a critical issue: how do we coordinate software development? At the same time they coupled the problem with a solution: use good communications to coordinate software development work.

Project Management since Garmisch

Software engineers and sociologists have been interested in the difficulties of software project management since the Garmisch conference.[3] They have expanded our understanding of what it means to manage a software project. Others have also proposed methods for managing the development process. In this section I review their research.

Brooks (1987) identifies four inherent properties of software that make it difficult to build: complexity, conformity, changeability, and invisibility. When software modules interact with each other they take the system into a different state of operation. Brooks measures complexity by the number of states that software can enter during run-time. As software gets larger the amount of elements increases. At the same time the amount of potential interactions between the elements usually increases exponentially. Software project managers find it difficult, if not impossible, to know all the possible states of a large system, let alone understand how they occurred. This makes it difficult to comprehend the system as a whole and extend it.

[3] A number of sociologists have been especially concerned with requirements elicitation. (See for example, Goguen and Jirotka (1994) and Quintas (1993))

Software must also conform to a variety of circumstances and needs. Brooks describes the origin of the conformity as,

> ...forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. (Brooks, 1987; 12)

Software developers must work with an environment that reduces the amount of potential actions that they may make.

People can easily change software because software is,

> ... pure thought-stuff, infinitely malleable. (Brooks, 1987; 12)

Because software is malleable, a developer can easily change it. Like hardware, software gets changed during development; however, software can also be changed once deployed. Also development does not often proceed from scratch; instead developers usually start with existing systems and adapt them to the new environmental circumstances such as a change in hardware, organizational policies, market conditions, or laws.[4]

Software is hard to visualize which makes it difficult to work with. Although developers can describe the flow of control or data in a system, no single visualization captures all the aspects of what the software does. Describing one piece of software using multiple visualizations can be confusing. Brooks claims that this creates problems for individuals trying to understand and develop software and talk about their work with others.

Brooks is most famous for describing the woes of software project management in his book *The Mythical Man-Month* (1975, 1995). In the book he describes his experiences of managing the development of IBM's OS/360. He also tackles the same question raised at Garmisch, how do you organize people in large software development projects, and what role does communication play?

He called the book *The Mythical Man-Month* to highlight a key management problem. He criticized a popular metric for measuring the effort required to build software, the man-month. The man-month is the sum of work one person can achieve in a month. For example, an eighteen man-month project could be built in the following ways: by one person in eighteen months, by six people in three months, or by nine people in two months. However, as Brooks succinctly points out,

> Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication among them. This is true of reaping wheat or picking cotton; it is not even remotely true of systems programming. (Brooks, 1995; 16)

---

[4] When development starts from an existing system, the old software is called legacy code.

Brooks statement about the communication required to build software has been confirmed by another group of researchers who studied software development in a large organization:

> Software development is not an isolated activity. Over half our subjects' time was spent in interactive activities other than coding, and a significant part of their day was spent interacting in various ways with coworkers. (Perry, Staudenmayer, and Votta, 1994; 45)

Software engineering researchers have developed "standards" around which various participants can organize and coordinate their work to try to reduce the problems created by communication (Pickering and Grinter, 1995). Some of these standards are not project specific such as, formal specification languages, software development life cycles, and formal process descriptions. Others must be engineered for the project at hand including, requirements documents, project plans, and testing plans. These standards have a technical purpose; to specify parts of the system, to guide the phases of development, and so forth. At the same time they provide shared definitions to all the project members reducing the need for communication.

Parnas and Clements (1986) recognized the dual purpose of these standard setting devices. They claim that even with the best intentions a rational design process will breakdown at times due to ambiguous and unknown requirements, changes during design, and human error among other reasons.[5] However, Parnas and Clements also cite reasons for acting as if the design process was followed and documenting all the steps,

> When an organization undertakes many software projects, there are advantages to having a standard procedure. It makes it easier to have good design reviews, to transfer people, ideas and software from one project to another. (Parnas and Clements, 1986; 252)

However, in a study of 17 different software development projects, Curtis, Krasner, and Iscoe (1986) found that these documentation strategies did not eliminate the need for communication. They observed that at the beginning of projects developers spent considerable time defining common terminology and creating informal channels for communicating project information.

Curtis, Krasner, and Iscoe also observed that knowledge about the application domain was thinly spread among the developers working on a project. Developers spent considerable time establishing a common understanding of the application domain and how the system should work. They discovered, like the participants at Garmisch, that requirements remain unstable throughout development. The requirements documentation often changed as a result of conflicts between different parties on the project.

Recently, software process research has attempted to reduce the amount of communication necessary by formalizing the process of software development and embedding that in systems (Curtis, 1995; Dowson, 1993). Software process researchers believe that the emphasis on

---

[5] Button and Sharrock (1994) verify and update this observation. In their study of a software engineering project, they noticed that the developers often used a CASE tool to document the process as it should have occurred, rather than use it to actually do the work.

documentation draws attention away from the real problem, developing software. The approach involves modeling the software development process and then building systems that implement these models. Some research systems have been built, but few have been integrated into software development practice yet.

2.5 A New Look at the Hard Problem

Software project management researchers have observed and noted the importance of communication and coordination in software development, but few have asked why it occurs. At one level the answer is obvious: developers need to synchronize their work with others and so must find out what their colleagues are working on. However, we can ask at a deeper level, why are communication and coordination necessary?

In this thesis I claim that dependencies between the different code modules create and reflect social dependencies between developers, managers, and software development organizations. Further, I will claim that developers must communicate and coordinate with each other to manage these dependencies. This thesis extends our understanding of project management, by providing an explanation of why developers, managers and organizations must coordinate to build software. This assertion is not completely new, and in this section I review observations other researchers have made about dependencies. My thesis will then provide a detailed explanation of how these dependencies manifest themselves, how developers and managers cope with them, and a framework by which to understand them.

At Garmisch the participants referred to dependencies obliquely in their discussions about software development. They recognized two different kinds of dependencies: those among code modules and those between software and all the other items that comprise a system. They observed that it was difficult to assemble the whole system from its parts because different code modules depended on each other and needed to be ordered to reflect that dependency. They felt that keeping different parts of the system synchronized — making sure that software worked with the hardware, and that the documentation matched the software — should be considered part of the development process. The participants treated these relationships as technical, links between artifacts that needed identifying and addressing. They appeared to assume that once recognized the problems created by dependencies would be easily resolved. This research demonstrates that even when recognized dependencies remain hard to manage.

Since Garmisch other researchers have commented on both the technical and social aspects of dependencies. Brooks describes dependencies in his discussions of the problems of software development. In his discussions of conformity he points out that software must align with the needs of institutions and other systems. He elaborates on this in his description of changeability:

> In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product. (Brooks, 1987; 12)

Sociologists like Woolgar (1994) and Whittaker and Schwartz (1995) also comment on the social aspects of dependencies that they found in software development. Woolgar says,

> From a sociologist's point of view, the requirements process will always involve the creation and maintenance of (often new) social relationships across social organisational (and sometimes institutional) boundaries. (Woolgar, 1994; 204)

Whittaker and Schwartz also note the role of dependencies,

> There are multiple dependencies within and between projects, necessitating careful sequencing of tasks, and frequent communication about progress. (Whittaker and Schwartz, 1995; 497)

While these authors have all observed dependencies, to date they have not been studied systematically. For example, Brooks mixes institutional levels of dependencies, such as those created by legal authorities, with those generated within a specific organization, like user demands. Whittaker and Schwartz, also capture organizational level requirements; however, they also point to another type of dependency, those that occur within a specific project, between developers working on the same software. The framework that I will propose categorizes these different kinds of dependency into clearer analytical units.

Finally, Scacchi (1984) identifies two sources of dependencies in his analysis of the social aspects of software project management. At the inter-organizational level, he situates software development in a context of legal and market forces that influence the systems built. He separates these from organizational dependencies, those factors that shape the development of software that come from inside the organization. I consider another level of dependencies, those that occur between two developers, the individual level.

2.6 Summary

Software engineering as a domain of research has been active for a short time, since 1967. In that time advances have been made, but many questions remain. This thesis aims to contribute to the collective understanding of one of those questions: how do we manage software effectively? That question being far too broad to be answered in one thesis, I have picked one aspect: how do developers manage dependencies?

Chapter 3

Configuration Management and the Coordination of Software Development

> Much of CM [configuration management] is concerned with controlling change: assessing the impact of a change before it is made, identifying and managing the multiple versions of items which a change generates, rebuilding derived elements after source elements are changed and keeping track of all the changes that are made to a system. Change is hard to manage because items depend upon each other. An apparently minor change to one element may propagate to items which depend upon it, directly or indirectly, so that consequential changes are needed throughout the system. (Whitgift, 1991; 24)

Software configuration management is the discipline of identifying components of a software system, putting those components together in the correct order, and controlling changes to the software during development. Although configuration management sounds simple, in practice people find it difficult. It is hard because identifying components of a software system, putting them together, and controlling change involves dependency management. For this thesis, configuration management is the part of software development to discover what dependencies exist, and how developers, managers and organizations manage them.

This chapter describes the emergence of the discipline of configuration management. The configuration management literature reveals that little is known about how these tools and policies support the coordination of software development in practice. However, research studies of other work settings suggest that individuals often need to coordinate their efforts to get the work at hand done. This literature reports mixed findings about the role of technology in facilitating the coordination of work. Finally site selection, methods, and theoretical perspectives used to gather and interpret data are described.

3.1 What is Software Configuration Management?

Configuration management practices and procedures evolved in military hardware systems development. During and after the second world war the demands for complex weapons grew dramatically. These technologies consisted of many sub-systems, often built by different organizations. For example, companies specialized in engines, guidance systems, fuselage, and so forth. Demands to build systems quickly and distributed development environments meant that companies did not keep accurate records of what had been assembled, and rarely did they

actually know exactly what was inside their technology, how it fitted together, and how it worked as a whole.

This came to a head in the late 1950's with what has become a legendary story within the configuration management community. As one configuration management book explains,

> This deficiency became apparent in the race for a successful missile launch in the 1950's. With time being critical, the promulgation of changes was accelerated to resolve incompatibilities among elements supplied by many supporting contractors. When a successful flight was finally made and the buyer, in the euphoria of success, said: "Build me another one," industry found themselves in the following circumstances:
>
> 1. Their prototype was expended (launched into trajectory).
> 2. They did not have adequate records of part number identification, chronology of changes, nor change accomplishment. (Samaras and Czerwinski, 1971; 15)

Incidents like this made military organizations realize the necessity of implementing two configuration management procedures. First, it was necessary to identify each component and the configuration of the components so that people would know what the system comprised, and how the developers had arranged those pieces. Second, it was necessary to track the changes made to each component, as well as alterations to the configuration of the system as a whole. If configuration managers did not track these changes, they would lose the ability to identify the components and understand the correct system configuration.

Configuration management continued to develop inside military environments. During the 1950's and 1960's the US. Army, Navy, and Air Force all developed configuration management standards. At first they were particularly concerned with aircraft and missile systems, but it slowly spread to other complex systems.

NASA also developed their own configuration management guidelines for space rocket development as the race towards manned space flight accelerated. In the late 1960's, the government, through the Department of Defense (DoD) begun to push for a standardization of the standards in place. They felt that different groups were reinventing the same standards. The result was a general standard for controlling systems development, MIL-STD 480. MIL-STD 480 was soon augmented by MIL-STD 483 "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs" (Department of Defense, 1970).[6] As yet software development had not been separated from hardware.

The DoD finally separated software from hardware when they introduced DOD-STD 2167 (1985).[7] The standard describes procedures for developing software and includes software configuration management procedures. Specifically it defines five aspects of configuration

---

[6] For a detailed treatment of the governments' efforts to standardize configuration management practices see (Samaras and Czerwinski, 1971).

[7] Although software had finally been separated from hardware, the basic principles of software configuration management did not vary from the more established hardware traditions.

management for software development: configuration identification, configuration control, configuration status accounting, handling and delivery of project media, and engineering change proposals. Together these characterize the goals of software configuration management, and so I will review each of these in turn.

Configuration identification involves identifying all the components in a software system. The components include: software modules, libraries, test suites, user documentation, requirements, specifications, and other artifacts generated during the development process. Not only must each component be identified, but each unique configuration of those components in the software system must be identified. Software systems may have more than one configuration; for example, different platforms may require slightly different variants of the product.

As the software evolves over time the individual components change. Configuration identification also includes versioning these changing components. Each time a developer changes any software component, a new version is made to record the differences. The composition of the configurations also changes during development as developers add and remove components. Configuration identification also encompasses versioning these different software configurations.

Configuration control involves managing the changes to the software. Changes come about because software requirements change, related hardware or software changes and so the system needs to be modified, and problems arise that need fixing. Configuration control involves creating a managerial review and approval process that prevents developers from changing the software autonomously. The intent is to maintain control over the evolution of the software so that software can be assembled.

Configuration status accounting involves documenting the details about the components, configurations, and changes. The accounting procedures originally consisted of creating and maintaining paper trails describing the systems evolution until the development of sophisticated configuration management systems in the mid-80's.

Handling and delivery of project media, and engineering change proposals, involve creating the appropriate documentation for the government client. Project media, documentation and code must be bundled and delivered in specific formats. One system may be spread across a number of contractors so this information helps the governmental agency assemble the software from the different parts. Engineering change proposals involve complying with certain standards determined by the specific governmental agency.

The procedures of configuration identification and control begin to reveal the importance of dependencies in development. Identifying configurations involves not only distinguishing different components, but describing how they fit together at compilation and build times. Change management activities track dependencies as they evolve during systems development. Although configuration management is a recognition of the importance of dependencies, the focus is on managing the technical aspects of dependencies.

3.2 Software Configuration Management Today

Software configuration management evolved inside the government contracting world. As a paper-based management discipline it was usually met with indifference by the academic software engineering community because it did not appear to provide any research opportunities. Configuration management issues were largely ignored by researchers interested in software project management as they concentrated more on managing the flow of work rather than the evolution of the system.

Today there are no configuration management journals or conferences. Although a number of books have been written on the topic, most of them orient themselves towards practitioners rather than researchers (for example, Babich, 1986; Compton and Conner, 1994; Whitgift, 1991). However, since 1991, there has been a workshop held once every two years, affiliated to the International Conference on Software Engineering. Slowly a group of researchers and practitioners have formed a group concerned with configuration management issues.

Configuration management became a research topic because software engineers began to explore the possibilities of automated support for software development. This automated support came as Computer Aided Software Engineering (CASE) tools that typically supported one aspect of the development process such as structured design, and Integrated Project Support Environments (IPSE's), aimed at providing an entire development environment.[8] Both streams of research presented researchers with opportunities to build configuration management systems. Some researchers built configuration management tools, including Revision Control System (Tichy, 1985), and more recently the Network Unified Configuration Management system (van der Hoek, Heimbigner and Wolf, 1996). Others worked on environments that placed configuration management at the center of software development work such as the Domain Software Engineering Environment (Lubkin, 1991).

At the same time researchers begun to building research systems, commercial vendors saw opportunities to develop and sell products. Today, the most technically comprehensive products come from vendor organizations, rather than from academic research environments.[9] Two forces have conspired to make configuration management a viable market.[10] First, the need to comply with standards has pushed commercial organizations to buy configuration management products. Second, the demand to create "open systems" has dramatically increased the complexity of software development.

Two new standards have recently begun to influence the way that commercial companies build their software. In 1987 the International Organization of Standards released their own quality

---

[8] IPSE's are also known as Software Development Environments. More recently there has been a blurring of the traditional distinctions between CASE and IPSE's with the introduction of integrated CASE (I-CASE). I-CASE provides a collection of individual CASE tools that together aim to support the entire development process.

[9] This is recognized by academics as well as commercial organizations. Academics build configuration management systems to answer focused questions about the role of technological support in software development.

[10] Some configuration management vendors have been around for some time. They sold their software to military organizations who needed help meeting their contractual obligations, or they sold rudimentary configuration management systems as part of an overall suite of development tools.

standards for products the ISO 9000 series. These series of quality assurance standards contain a configuration management standard. Today, many European companies, governments, and European subsidiaries of American owned companies, must buy software from companies certified as ISO 9000 compliant.

In the United States another standard, primarily aimed at the military contracting world, has gained importance. The Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI) is a standard for measuring how well a company builds software. The CMM consists of five levels. At level 1, a company develops software chaotically, they have little control over how the process occurs, and cannot repeat it. At level 5, the organization has an optimized, repeatable process, and when they occasionally make a mistake they can retreat back to a working product quickly and learn from the errors to avoid repeating them. They can also accurately estimate the time needed to build any software.

To date two companies have reached level 5 and most organizations operate at level 1 (Gibbs, 1994). To reach level 2 the CMM mandates that the organization has a configuration management process, among other things. The model has become important because the U.S. Air Force has mandated that by 1998 all companies competing for contracts must be at level 3 or above (Gibbs, 1994).

At the same time open systems have created a demand for configuration management tools. Software product development has transformed from a proprietary to an open systems industry in the last ten years. Once, many software companies developed applications that ran on their own hardware and networks. Today those organizations must provide applications that work with a variety of operating systems and hardware built by other manufacturers. For example, an application could be expected to run on six different platforms and be compatible with three commercially available databases. Thus the development organization may need to maintain up to eighteen different variants of a single application. Most software product organizations find it hard to keep their development environment ordered. Questions about the products being built often come up: which piece of functionality belongs to what release, which platform requires a certain piece of code, what part of the documentation needs altering to make it compatible with this release, and how can the variants be tracked.

The trend towards open systems has created a demand for configuration management systems because the identification and change aspects of configuration management have outgrown the paper-based methods of accounting. Identification now involves tracking many variants of the application. Making changes has also become more complex. Some changes need to be implemented across all platforms and substrates; for example, a new application functionality. Other changes may be localized to a specific platform or substrate; for example, arising as a result of a change in the underlying technologies.

3.3 Configuration Management Systems

Configuration management systems aim to provide automated support for configuration management work. First generation configuration management tools used a library metaphor of "checked-out" and "checked-in" states to control changes to software. To make any modifications to a software module, developers had to check out the code. When a developer checked a module out, the tool made a new version of the code and prevented others from checking out the same software. When changes had been completed, the developer checked in the code. A checked-in module was stable and usually working. Other developers could read and execute it with their own modules. By checking-out and checking-in code, developers created successive versions of the module that the system stored. Code versioning created stability during development by facilitating backtracking to older versions if necessary and preventing developers from overwriting the work of others.

However, first generation configuration management tools had two disadvantages. First, they only worked for code. However, software systems also contain libraries, test suites, makefiles, and documents that change during development. Modern configuration management systems use a database to store all the artifacts that make up a software product. Second, the checked-out state turned out to be very limiting because it prevented others from changing the same module at the same time, which slowed down developers' ability to get their work done. Modern systems solve this problem by allowing two or more developers to work on the same module at the same time and then merge their changes together.

Modern configuration management tools support three layers of functionality on top of the versioning facility (Caballero, 1994). The configuration control layer maintains information about the artifacts that form a software product. It knows which versions comprise a specific system and how they relate to each other. This layer allows developers to pull together all the software artifacts that comprise a specific variant of the software using a make-like utility. It also lets developers recreate both previous and current releases of any software stored inside the configuration management data repository.

The process management layer provides a "life cycle" for each type of artifact stored in the system. A life cycle consists of a number of states. For example a typical life cycle for a software module consists of the checked-out, checked-in, quality-tested, and released states. While the developers are most concerned with the checked-out and checked-in states, testers of the software use the quality-tested state to signal that a particular version of a software module has passed rigorous system testing.

Finally, the problem reporting layer supports bug and enhancement tracking. Modifications to the artifacts in the system occur as a result of problems with the functioning of the system or enhancements requested for future products. The problem reporting layer provides a way of linking the bug reports or enhancement descriptions to the changes themselves. Modern configuration management tools either have built in process management and problem reporting, or provide the necessary connections to allow users to build it themselves or buy another off-the-shelf system and integrate it into the configuration management tool.

3.4 Explanations of how Configuration Management Works in Practice

Little has been written about configuration management generally, and even less has been said about how configuration management happens in practice. The literature can be divided into three categories: prescriptive visions of how to implement configuration management procedures, technical literature about the role of configuration management systems, and a few articles that suggest what realities of practice might be.

Most books about configuration management explain how to implement policies and procedures, and occasionally tools, for practitioners (see Compton and Conner, 1994; Whitgift, 1991). The authors describe the difficulties of software development: challenges of communicating change, of organizing multiple people to build a single software system, and knowing what any system contains at any given time. Having discussed the problems they suggest how configuration management reduces or eliminates them. As Bersoff, Henderson, and Siegel (1980) say,

> SCM [Software Configuration Management] … is defined as the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle. (Bersoff, Henderson, and Siegel, 1980; 20)

The authors describe the main functions: identification, status accounting, and change management; however, they rarely mention the environment in which their configuration management practices and policies will function. They concentrate on defining those practices instead.

When these authors attempt to deal with potential difficulties in the environment, they focus on specific personality types. Several of the books have attempted to classify the different types of problem people; for example, Babich identifies the renegade programmer as:

> They know that the configuration management procedures (the "bureaucracies") are a waste of time, not to mention an affront to their individuality, creativity, and constitutional rights. They are going to do what they believe is best regardless of what you tell them. (Babich, 1986; 94)

He ends up cautioning potential configuration managers to act diplomatically with "difficult" developers.

Compton and Conner (1994) take these characterizations further as they describe the guru,

> Gurus must have things done their way to remain Gurus; compromise is not in the creed. In their formulation of the universe, Gurus sit next to (and advise) the god of software, and all access is through them. This mindset is rarely suitable as the basis of a global SCM policy. (Compton and Conner, 1994; 101)

Their characterizations continue, the cowboy, essentially a nice programmer who leads the crowd and often ends up disobeying software configuration management procedures and the loner unused to working in teams. Their solution is to discipline the offending members of the team. Explanations that identify obstinate programmers have some foundation in real situations; however, they do not account for the times when developers find it difficult or impossible to implement configuration management procedures in practice.[11]

Recently, a number of technical publications have noticed the trend in automated configuration management systems. Instead of describing configuration management practices they concentrate on the kinds of system functionality available, the uses of those features, and the merits and disadvantages of specific systems (see Caballero, 1994; Fromme, 1994; LeBlang, 1994). Again these articles rarely provide any information about the difficulties of implementing systems in specific software development contexts. In fact they usually like to report on unproblematic cases, organizations that embraced configuration management systems, and found nothing but benefits.

However a few authors have commented on the challenges of implementing and using configuration management systems. Susan Dart (1992) observes that managerial and political issues play a critical role in the adoption of tools and practices. Dart believes that upper levels of management must be ready to manage technology transition by persuading people to use configuration management systems, customizing the tool to fit into the existing work practices, and recognizing that changes arise from the adoption of any new technology. Management must also make choices about whether they should buy tools of the shelf, or build and maintain their own. The political issues involve the mandated use of configuration management by the Federal government through standards like the CMM.

Dart's work emphasizes the adoption of configuration management systems, and their associated practices, by an organization. However, she also makes an important observation that configuration management systems do not simply affect the work of individual developers in an isolated way, but impact the entire organization. Babich (1986) also identifies this,

> On any team project, a certain amount of confusion is inevitable. The goal is to minimize the confusion so that more work can get done. The art of coordinating software development to minimize this particular type of confusion is called **configuration management**. Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. (Babich, 1986; 8, bold in original)

Configuration management systems are a form of groupware technology, and as well as affecting individuals they require organizational commitments to adopt and use. Recently, Nix (1994)

---

[11] Although we can excuse their limited solutions as a lack of understanding of the complexities of software development, these suggestions also ignore something that they should be more familiar with. Given the history of configuration management as a paper-based accounting discipline, configuration managers often have a weak position within their organization. As well as representing tedious paperwork, their career trajectories remain more uncertain that those of developers. These books often assume that the configuration manager has some authority over the developers, but in reality this may not be the case.

drew a parallel between configuration management and groupware, by claiming that tools acted as a communications hub for developers working on common software. However as Grudin (1994) observes of groupware systems generally they receive less attention and visibility than systems used by everyone in the organization, but they still need support of management during the adoption phase if they are to succeed. It is these concerns that Dart tries to address in her work through raising the consciousness of management to these issues.

Davies and Neilsen (1992) have examined configuration management in one setting.[12] They conducted their study at the Information Technology Centre (ITC) of a university in Queensland, Australia, using qualitative methods to gather and analyze data. They found that the model of rational actions assumed by configuration management policies, and reflected in the documentation that ITC generated, did not accurately reflect their everyday practices. Although their informants completed the required documentation it did not necessarily imply that they had resolved their configuration management difficulties. Instead they hid their configuration management difficulties behind the completed documentation.

Rather than starting with normative vision of how configuration management should occur, this research describes the configuration management practices used by developers and organizations. It shows how these practices have technical underpinnings, in the dependency relationships between pieces of code. At the same time it focuses on the social practices and conventions that help to manage those dependencies, and the role of configuration management systems in supporting dependency management.

3.5 Computer Support for Groups

Although the configuration management literature contains few reports of practice, another body of research provides important background for this work. Researchers who participate primarily in the Human Computer Interaction (HCI) and more recently, Computer Supported Cooperative Work (CSCW) communities have been interested in how people coordinate activity. Much of the sociological work has concentrated on looking at collaborative work practices, and the role of technology within those practices (for example Heath and Luff, 1991; Suchman, 1992; Rogers, 1993). This literature provides a number of insights for this research.

Researchers have noted that the dependencies between different work functions imply relationships between people. For example, individuals make their work visible to others by speaking out loud and arranging papers so others can see them (Heath and Luff, 1991). They also monitor the work of their collaborators to learn about events that may have a bearing on their own work (Hughes, Randall and Shapiro, 1993). The arranging of papers and the arrangement of people also provide important spatial cues that individuals use to interpret the current state of events and align their work (Anderson and Sharrock, 1993). This research

[12] They do not state whether the participants were using configuration management for software, hardware, or documentation.

demonstrates that this is also true in software development, where technical dependencies create and reflect social relationships among people and organizations.

Ethnomethodological sociologists have described the ways that people work together to establish a common understanding, an account, of their work. The accounts individuals produce often help others to know the current state of work (Suchman, 1983; Sharrock and Anderson, 1993; Button and Dourish, 1996). Davies and Neilsen (1992) found accounting activities going on in their study of configuration management practices. This research shows that both practices and tools provide accounts of work that developers use to understand what the current state of development is.

In practice work often differs from the prescribed plan of action (Suchman, 1987). This happens because work takes place in a dynamic environment, where unpredictable and unplanned events occur. This observation suggests that in practice configuration management work may differ from the planned procedures.

More recently, CSCW researchers have begun to explore the problems and issues of groupware systems in organizations (Grudin, 1988; Orlikowski, 1992; Bowers, 1994; Ackerman, 1994). The systems that they studied include meeting schedulers, Lotus Notes™, a network of CSCW applications, and organizational memory. These researchers have reported on a number of general challenges that users of these tools face in trying to make them work. Among these issues are: the relationship between people's understanding of a technology and its use, the mis-matches between who does the work and who gets the benefit, and clashes between existing organizational structures and the use of groupware. Other researchers (Perin, 1991; Pickering and King, 1995) have shown that inter-organizational associations, such as professional communities, influence the adoption and use of groupware systems.

Software configuration management systems provide another venue to study these issues in rich detail. The developers of software configuration management systems have been relatively isolated from the groupware community; as a consequence, the tools differ from more "traditional" groupware systems. Many traditional groupware systems, like electronic mail, video-conferencing, and media spaces, support collaboration by providing mediums for communication. Configuration management systems try to support collaboration by providing information about the current work-in-progress and what other developers are doing, as well as providing models of how software development proceeds. Configuration management systems are similar to workflow systems because they structure software development activities. Workflow systems have been a topic for debate in the CSCW community, but few people have examined their use in organizations (Suchman, 1994; Winograd, 1994). An empirical study of a technology that supports collaboration by providing information to help developers coordinate with each other may help build a more comprehensive picture of computer-supported work.

3.6 Methodology: Qualitative Research and Site Selection

Although previous research in software engineering, software project management, configuration management, computer supported cooperative work and human computer interaction provides useful pointers, the question of how developers manage software dependencies remains unanswered. In the absence of previous systematic studies of dependency management it was impossible to generate testable hypotheses. Instead a qualitative research strategy was chosen because it supports exploratory research (Marshall and Rossman, 1989).

Quantitative sociology remains the dominant methodological approach to understanding human society. However, qualitative sociology, despite being marginalized at times, has a tradition beginning with symbolic interactionism in the 1920's.[13] Symbolic interactionism placed the actor at the center of the phenomena being studied.

> We want to know what the actors know, see what they see, understand what they understand. (Schwartz and Jacob, 1979; 7)

This study was concerned with how developers and organizations understand and cope with dependencies in practice. Qualitative methods with their focus on the actors and their concerns focused this research on the problems of dependency management. The participants in this research found dependency management a difficult and time-consuming task.

This thesis describes dependency management practices at three organizations. Although I visited other sites, I chose these three because they illustrate the main points of this theory of dependency management. I briefly describe the sites and my reasons for selecting them, as well as describing the methods used to gather and analyze the data. Each site is described in detail in the chapters that follow.

Tool Corp. is a small development organization, with about 150 employees when I arrived in the middle of 1994. They build a configuration management system that they sell on the open market.[14] I began my thesis work at Tool Corp., and they provided me with my initial exposure to the world of configuration management. I selected Tool Corp. for three reasons. First, they had configuration management practices in place and used their own system in-house. Second, they managed to release and sell software successfully. Finally, I gained acceptance into the company very easily. I approached the Vice President responsible for overseeing the development of their product. He welcomed me into the organization by providing an office and access to all of the developers. He also seemed comfortable with the idea of having me stay there for an extended period of time.

During the four and a half months that I remained inside the organization, I used a variety of methods to gather data: participant and non-participant observation, semi-structured interviews, informal interviews, and document analysis (Bernard, 1988; Jorgenson, 1989; Lofland and Lofland, 1984). At the same time I attended meetings in the organization, followed a USENET group relating to configuration management, and as I became more familiar with the terminology

---

[13] Some techniques of qualitative sociology have roots in anthropology such as ethnography, so the tradition of studying individuals in the field dates back even further.

[14] I discuss the problems and benefits of studying a vendor of configuration management tool and their usage of the tool that they built in chapter 4.

of configuration management, conducted broader searches for literature. I describe these methods in the next section.

The data from Tool Corp. grounded this study and provided much of the conceptual framework. However, as a configuration management system vendor they emphasized the importance of configuration management practices and technologies. In qualitative research, researchers should not necessarily pay attention to the anomalies of their sites, as each site happens to be different. In this case the anomalies were extreme, and Tool Corp. also offered me connections into other organizations. These two factors made further studies desirable and possible, and the data that I gathered from those sites radically altered, and improved, my understanding of dependency management.

The other two organizations that appear in this research have connections to Tool Corp. Computer Corp., is a large computer manufacturer in the Bay Area. As well as having guaranteed access and sharing the use of Tool Corp.'s product, I chose this site for two other reasons. First, the organization builds software in a similar context to Tool Corp. Essentially they make products that they sell on the open market, although for special customers they provide customized solutions. The mix of primarily product development with a little contracting provides a balance between Tool Corp. and the third site. The second reason for selecting Computer Corp. was the scale of the operation. Computer Corp. has approximately 700 developers working on the same large set of software products, their software solution. The size of the company altered the strategies of coping with dependencies, so this site added some key elements to my understandings of dependency management.

Computer Corp. recently adopted Tool Corp.'s product. In this study I spent three days interviewing and observing people in the organization. The primary data source comes from semi-structured interviews that I conducted with a cross-section of the organization. I also gathered background information about the company, using the World-Wide Web.[15]

The final site proves the most problematic in some ways, and the most interesting in others. Contract Corp. is a software development organization that specializes in contract software development work. This creates a contrast among the three sites, Tool Corp., a product developer, Computer Corp., primarily builds systems for the open market, but also provides some customized solutions for special customers, and Contract Corp. that only engages in contract work. A second reason to visit Contract Corp. was that while the organization as whole does contract work for a variety of commercial and governmental operations, the site I visited had recently completed a military project. This provided me with the opportunity to visit a site and gather data about dependency management in a military contracting setting.

However, as a foreign national without security clearance, it also meant that my access was restricted to a senior project manager and a configuration manager. While they both talked to me for several hours, in semi-structured interviews, they may have been much more guarded about

[15] As well as visiting the company's World-Wide Web (WWW) site I searched the Internet for references to the organization. I found annual financial reports, news releases concerning the company, and information about the company's competitors. Although the web site of the company tended to emphasize the products that they sold, newspaper reports from newspapers on-line provided other information about the organization.

their opinions than others. It was also harder to get background information about the company itself. Despite these limitations, Contract Corp. provides an important data point. I wanted to test the emergent theory in another setting, see what carried over, and what needed adjusting to accommodate the differences between military and commercial configuration management.

Data gathered at four other sites does not appear in this write up. Two other sites also worked on commercial products, very much in the style of Tool Corp. The other two worked in military and quasi-military contracting settings, like Contract Corp. These sites, discussions on comp.software.config-mgmt, and conversations with my colleagues developing software do not appear in this thesis, but the influence of these other data sources remains in the framework. These sites, people, and colleagues are the silent partners of Tool Corp., Computer Corp., and Contract Corp.

3.7 Methodology: Data Gathering and Analysis Using Grounded Theory

At Tool Corp. I conducted a long-term in-depth study of the participants' daily software development activities. I focused on their configuration management practices and the tools they used. In the beginning I used non-participant observation strategies to collect the broadest data possible. This consisted of observing the informants at work, and maintaining a diary of happenings, as well as thoughts and feelings about the site (Lofland and Lofland, 1984). I also learned to use their product, which provided a hands on opportunity to explore how developers used the tool. I kept the e-mail that I sent from the site to friends and colleagues, as records of my reflections on the study and site. These non-participant observation strategies sensitized me to the environment, and helped me to begin to interpret and make sense of the data. They also provided a permanent reminder of my initial reactions to the site, which proved useful when I became used to Tool Corp.

At Computer Corp. I conducted some observation of people in the configuration management group advising developers about practices and technologies. I also attended a class along with other developers where they learned about the new product, Tool Corp.'s tool, that their group would be shortly adopting. This provided an interesting opportunity to learn about other organizations' perceptions of the product. It re-sensitized me to the difficulties of understanding how the tool worked and the way that it organized software development practices.

As I became more familiar with Tool Corp.'s development environment, and the participants grew aware of my skills, I was invited to help with the development effort. I did no coding, but conducted usability tests and multiple user testing of the product under development. I also participated in the company's retrospective analysis of the software development life cycle, conducted whenever a new product has been shipped.

The participant-observer method has a history in both anthropology and sociology as a mechanism to get at the subtleties and sometimes hidden meanings held by the informants (Jorgenson, 1989). However, as the researcher becomes increasingly used to the environment they risk a loss of objectivity, often called "going native." When a researcher goes native they

lose their ability to interpret the events going on around them, by taking them for granted as the participants themselves do. Strauss and Corbin (1990) provide a number of ways to help the researcher maintain a theoretical sensitivity to the environment that I adopted during this study: asking myself questions about what was going on around me, withdrawing from the field after several months to reflect on my experiences, and building competing interpretations of events.

I used unstructured interviewing strategies to gather data at the first site (Bernard, 1988). Unstructured interviews have little if any interview guide, as the purpose of the interview is to find out more about the person and their concerns. The interview is guided by the topics that the informant wants to discuss. I tried not to steer the interview in any direction as I wanted to learn about how the informant felt about issues of their choice, and find out how they put events and objects together in meaningful ways. Unstructured interviews allowed me to gather a large volume of data about a broad range of topics at the beginning, and during later phases of the study, became useful check points for any missing information.

I also conducted semi-structured interviews at all three sites. These involved an interview protocol and were held in a formal setting. I scheduled the interviews, taped, and transcribed them. I used an interview guide, although I frequently deviated from the guide either willingly because the informant raised important and new questions, or reluctantly, as the informant moved the conversation away from topics of interest to me. The interview guide changed over time as my understanding of dependency management evolved. However, the opening question always remained the same:

> What do you do here at X organization?

I also was sure to ask two other questions:

> How do you do configuration management here?
>
> What automated support do you have here for configuration management?

These questions usually led the informants to describe the processes within the organization. This helped them to relax into the interview. During these descriptions I took notes about important parts in the process, where they seemed to be describing dependency management, and I used probing techniques to gather further data about those specific topics.

To triangulate the data among the three sites I always asked some common questions. To triangulate the data within each site I took themes and concepts that emerged from one interview and asked about them in subsequent interviews. Finally, I used the silent partners to further confirm the hypotheses developed from the research.

I used grounded theory, a qualitative data interpretation strategy to analyze the data gathered from the sites (Glaser and Strauss, 1967). Grounded theory offered two advantages. First, several books and papers have been written about grounded theory and they provide rich details about how to operationalize the concepts. Grounded theory was originally proposed by Glaser and Strauss in 1967. Glaser and Strauss have written books explaining how to conduct grounded

theory studies since then. I followed guidelines proposed by Strauss (1987) and Strauss and Corbin (1990) in this study.

Second, grounded theory suits this study because it meshes perfectly with the theoretical perspective of articulation work and social worlds that I used to focus the later stages of data collection on coordination issues. Grounded theory is not entirely inductive and can leverage from existing theoretical bases provided that the theory it relies on has also been developed in a grounded manner (Strauss, 1987). Articulation work and social worlds were the two theoretical approaches used in this research to organize the dependencies that were discovered in the field. I describe both of these perspectives and their relevance to this work in the following sections.

Grounded theory calls for a continual cycle between data gathering and data analysis. The researcher continually tests their understanding by gathering more data that confirms, contradicts, or extends the theory being developed. Although I describe data collection and analysis as sequential stages, they happened in cycles. Strauss and Corbin call this process of testing the developing research "theoretical sampling."

The development of grounded theory consists of three main stages: open coding, axial coding, and selective coding. Open coding consists of reading through data such as interview transcripts, observational diaries, and documents. The aim of open coding is to find categories that explain the behavior described in the data. These categories initially have names, and properties that vary on certain dimensions. The next step, axial coding involves developing these categories, finding the conditions that lead to their emergence and the consequences of their occurrence. During selective coding a researcher picks one category as the core category, the category that forms the center of the theory.

The whole process ends when the researcher reaches a point of theoretical saturation. Theoretical saturation occurs when analysts get nothing new from data that they gather. When the theory is complete, data gathered simply fits into the existing theory rather than extending it or altering it.

3.9 Theoretical Perspective: Articulation Work

Strauss defines articulation work as follows:

> First the meshing of the often numerous tasks, clusters of tasks, and segments of the total arc. Second, the meshing of efforts of various unit-workers (individuals, departments, etc.) Third, the meshing of efforts of actors with their various types of work and implicated tasks. (The term "coordination" is sometimes used to catch features of this articulation work, but the term has other connotations so it will not be used here.) (Strauss, 1985; 8)[16]

[16] The total arc Strauss refers to "consists of the totality of tasks arrayed both sequentially and simultaneously along the course of the trajectory or project." (Strauss, 1985; 4)

Two studies added important aspects to Strauss's definition of articulation work. Gasser (1986) describes a setting where the participants used technology in their work. He described different strategies of aligning, fitting and adjusting work that participants engaged in to accommodate the computer systems they had to use. These strategies form a part of the articulation of modern work, work that involves computer systems. Gerson and Star (1986) observed that articulation of activities may only resolve things temporarily. In their study of an insurance organization, they note that articulation of work may resolve a coordination problem temporarily, for this specific instance, but if the circumstances arise again then the solution may have to be negotiated anew. Gerson and Star emphasized the on-going nature of articulation work.

Strauss expanded the definition of articulation work later, to include what he called the articulation process,

> The overall process of putting *all* the work elements together *and* keeping them together represents a more inclusive set of actions than the acts of articulation work. (Strauss, 1988; 164, italics in original.)

Articulation work is the coordinating and negotiating necessary to complete the work at hand. Software developers primarily work on designing and building software systems. However, as Bendifallah and Scacchi (1987) point out, as software developers design and build software they must also engage in forms of articulation work. Configuration management systems attempt to support some of this articulation work electronically.

Schmidt and Bannon (1992) have applied the concept of articulation work to the research problems in the computer supported cooperative work (CSCW) community. They describe how individuals engage in articulation work as part of their daily routines. They say:

> However in 'real world' cooperative work settings … the various forms of everyday social interaction are quite insufficient. Hence articulation work becomes extremely complex and demanding. In these settings, people apply various *mechanisms of interaction* so as to reduce the complexity and, hence, the overhead cost of articulation work … These protocols, formal structures, plans, procedures, and schemes can be conceived of as *mechanisms*… And they are *mechanisms of interaction* in the sense that they reduce the complexity of articulating cooperative work. (Schmidt and Bannon, 1992; 18-19, italics in original)[17]

Examples of these coordination mechanisms include plans and standard operating procedures. These mechanisms supplement forms of social interaction like e-mail, video-conferencing, and other forms of communication.

From experiences of managing software projects, configuration management specialists developed computer systems to support configuration management. They did not build systems

[17] Since this paper Schmidt and his colleagues have further defined mechanisms of interaction for articulation work as coordination mechanisms (Simone, Divitini, and Schmidt, 1995).

that would increase the communications bandwidth, such as e-mail, for two reasons. First, in large development teams communication paths cannot support all the articulation work necessary to get work done. Second, coming from the software engineering community, configuration management specialists are used to, and comfortable with, formal approaches to resolving coordination problems (Pickering and Grinter, 1995). Instead of building systems to increase the communications bandwidth they embedded coordination mechanisms into a configuration management tool.

Each of the layers of a modern configuration management system attempts to support the coordination of software development. The check-out/check-in layer coordinates the day-to-day work of developers as they develop modules. The configuration control layer allows developers and managers to routinely gather the work of the entire development team into one product. The process layer synchronizes the activities of various groups involved in design, such as quality assurance and development. Finally, the problem tracking layer coordinates the definition of problems with the actual changes made to the code itself.

Configuration management practices and technologies provide an opportunity to examine the articulation of software development work. Unlike previous work that has connected articulation work to computerization, this study makes technology a point of articulation. This thesis shows how these systems shape and reflect software dependencies, and what limitations configuration management tools place on the articulation of those dependencies.

Strauss's distinction between articulation work and the articulation process appears in the data gathered. The articulation of work among developers as they work on individual modules was separated from the work that teams of developers had to do. In the three data chapters that follow individual dependencies focus on the articulation work that developers do. Group-level dependencies focus on articulation that teams and organizations have to do as a whole, or the articulation that goes on between different teams. Although articulation work focused data gathering on dependency management, as a perspective it did not explain one set of dependencies that exerted huge influence on software development, so I turned to social worlds.

3.10 Theoretical Perspective: Social Worlds

During the course of this research dependencies among different software development organizations emerged in the data. These inter-organizational dependencies impact people's lives, changing their priorities and providing them with new working arrangements. However, the theory of articulation work, while capturing the essence of those negotiations within a single organization, did not seem to provide an adequate explanation of these inter-organizational dependencies. Social worlds, particularly as described by Howard Becker (1982), offered insights into the character of dependencies that sustain software development worlds.

In his book, Becker explains how people often view art as an individual activity. The artist paints, the poet writes, the singer sings, and the pianist plays. However as Becker explains, art is a cooperative activity:

> Painters thus depend on manufacturers for canvas, stretchers, paint, and brushes; on dealers, collectors, and museum curators for exhibition space and financial support; on critics and aestheticians for the rationale for what they do; on the state for the patronage or even the advantageous tax laws that persuade collectors to buy works and donate them to the public; on members of the public to respond to the work emotionally; and on the other painters, contemporary and past, who created the tradition that makes the backdrop against which their work makes sense. (Becker, 1982; 13)

In the quote Becker explains how artists depend on both consumers and producers to support their work. This research reveals that in software development worlds both production and consumption dependencies exist. This work shows how these inter-organizational dependencies impact the organizations in these social worlds.

Becker elaborates on this point in his discussion of conventions. Conventions are the social arrangements necessary for this network of collaborators to work together. The network has few, if any, formal boundaries. The participants do not work for one single organization. They may only be partially bound by laws and other governmental regulations. However, because they depend on each other, they must establish and maintain conventions that allow them to interact with each other to their mutual benefit.

Becker defines these conventions as those mechanisms that allow the participants to interact, but hastily points out that they do not constitute immutable laws. Conventions are agreements between people that have come to represent the customary way of acting. Like Gerson and Star's observation about articulation work, these conventions may be re-negotiated every single time, or they may gradually change over time, or remain stable and then suddenly shift. He also notes that conventions are interdependent so that if one changes, others must often change as well.

Conventions in software development worlds do not have the same grounding in tradition as those in art worlds. However, conventions shaped by market forces, the government, and other communities of practice do influence software development worlds and the people building systems. Social worlds provide an explanation of the dependencies that influence software development but do not come from within the organization in which the development is taking place.

3.11 Summary

The rise of configuration management policies and subsequently configuration management systems reflects a growing concern within the software development community about the difficulties of managing the relationships between different components of a software system. This thesis extends that understanding by emphasizing the importance of both the technical and social aspects of those dependencies and providing some insights into how people manage them in practice. Studies of practice especially within the HCI and CSCW communities, suggest that

work creates social relationships among people, and indicates that technology can play a role in supporting that collaboration. However, little is known about the kinds of relationships that developers and organizations create and maintain during the development of software.

This chapter also focused on research methods and perspectives. Grounded theory is used to gather and analyze data. Articulation work focused on the interactions between individuals in the course of their everyday work. It revealed dependencies that developers must manage to build software. The theory of social worlds captures the situated context of software development, the fact that software development organizations depend on other organizations to guide and shape their development process. The next three chapters introduce the sites studied and describe the dependencies that developers, managers, and the organization manage as part of their routine software development activities.

Chapter 4

Case 1: Experts Using Configuration Management Tools Still Need Help

> You can never separate the two (design and marketing). It doesn't matter how great the car looks if the engine is broken, and they could be very technically advanced engines which break continuously, and this is fundamentally what happened. (8: 1592-1594)

> Knowing what the rest of the people on your project team are doing, well it helps there because you can kind of see at the data side. Do you know what their intentions are, what they going to do, the areas they are going to focus on, no. The [configuration management] system doesn't know anything about the future, it knows a lot about the past. And something about the present, but you know it's very hard to pick the present from a snapshot. (13: 2259-2262)

This chapter describes the results of the research conducted at Tool Corporation, a small development company that builds and sells a configuration management tool on the open market. The chapter begins by introducing the company and the software that they build. The next three sections of the chapter describe the three levels of dependencies found during the research study: individual, group, and inter-organizational. The research yielded a number of dependencies at each level. The technical and social aspects of each dependency are described and the strategies that developers and Tool Corp. uses to cope with each dependency is discused.

4.1 Welcome to Tool Corporation!

Tool Corporation is a small software development company with their headquarters in California. The company has existed since the early 1980's however they only began focusing exclusively on developing configuration management systems in 1986. By 1994 they had established a presence in the configuration management systems market with their own tool, and had developed several versions of the product.

In the nineties Tool Corp. has grown substantially in size, their average growth figure varying between 100%-200%. This growth can be attributed to the transformation of Tool Corp. from a start-up company to an organization that has an existing customer base, products, and the potential to capture and maintain a significant share of the configuration management systems market. As such Tool Corp. was growing marketing, sales, and services operations rapidly, as well as expanding into foreign countries.

The CM tool market that Tool Corp. competes in has grown rapidly in the last few years. Recently, Ovum, a London-based firm of consultants interested in analyzing the growth of various computer-related markets, predicted that the market for configuration management technology would be worth $1 billion worldwide by 1998 (Ingram, 1994). As described in the previous chapter, this is being driven by standards like the CMM and ISO 9000 as well as the push towards open systems. In a search for solutions, configuration management systems, offer comforting reassurance that ordered software development can be achieved.

Today, the configuration management systems market is dominated by an oligopoly of vendors. While there are some companies in this market that have been building systems for many years, the majority of the organizations in this oligopoly are young companies, reliant on venture capital, seeking to turn initial profits, make initial public offerings and so forth. The focus of the market has shifted rapidly from UNIX oriented tools to a trend towards supported mixed development environments, especially a combination of UNIX and PC machines. Recently Microsoft entered the configuration management systems market when it purchased a small configuration management system vendor, but it is unclear whether this will have any significant affect on the market.

Tool Corp. builds and sells a high-end configuration management system on the open market. They use their own configuration management tool internally to help them control the development of the next versions of the product. During my time there, the company was preparing to release a new full release of their tool and a new point release for a variety of hardware and software platforms. The development group that studied consisted of 14 people when the study began, and 3 testers, as well as the manager. It grew over the course of my study to approximately 18 developers. After the release of their new products Tool Corp. reorganized the development group. They created a new tier of management and special software development roles including architect and build manager.

The shift from building strictly UNIX based configuration management systems to building a PC client to their system marked an important time for Tool Corp. When I arrived at Tool Corp. the developers all had UNIX workstations and several of them had two on their desks. Their last public release of their configuration management system ran on a variety of UNIX based platforms such as; HP, Sun and DEC. However, Tool Corp. along with its competitors began to shift more seriously towards PC development during my time there. By the time I left all the developers had PC machines on their desks.

The first site in my study was a development division of one CM tool vendor that I call "Tool Corporation," that competes in an oligopoly for this market. Specifically I studied how the developers responsible for building the CM tool use their CM tool to manage their work. The group consisted of 14 members, including the manager, and software testing group, who also use the tool in their daily work. Because the developers use the CM tool to build the latest version of CM tool itself, they are experts in using it.

Obviously, studying expert users of the CM tool affects the conclusions that I can draw, but it also offers several advantages. By studying a group of experts who have used the technology for some time I did not find problems of adoption reported in other studies (for example, Grudin,

1989; Orlikowski, 1992; Bowers, 1994). Second, even though the developers know their product extremely well they still had to manage the same software dependencies as other development groups. I used participant and non-participant observation techniques. I also conducted 20 semi-structured interviews and approximately 80 informal interviews.

4.2 Individual Dependencies at Tool Corp.

Life at Tool Corp. revolves around the product that they build and use in their development process. This tool occupies their attention, as a way of organizing the development life cycle, as design decisions that they must make, as their livelihoods. The tool also helps the developers to cope with the dependencies that they experience in their work. In this section I describe the dependencies that they encounter and the role of the technology in managing them.

Parallel Development Dependencies

The developers call the times when more than one person has the same module checked out, "parallel development." This happens when different developers have changes that require them to work on the same module. The tool supports this by allowing both the developers to check-out copies of the module and make their changes.[18] The tool also provides a merging facility, which lets developers integrate their changes with those made by their colleagues. Despite the automated support that the tool provides the developers still try to avoid parallel development, because it creates dependencies between them, that take time and energy to resolve.

Developers often explained that parallel development represented weaknesses in the product:

> Well I try to avoid it, I grumble about it, to me it's out there, it happens in our company and in others, but it seems to me that if there's better management and better decomposition of problems then should be avoided. ... Number 1 solve it by keeping things separate as far the units of work, the resolutions of work, which in our case is source files, and number 2 when you go about assigning this work you could try and assign common problems to the same person so they are not doing parallel development. (3: 447-456)[19]

> Anytime I see a parallel occurring at all on the same project to me that's a flag that this module is doing too much. And perhaps the module itself needs to be broken up ... usually this set of functionality belongs to me, other people working on the

[18] Other tools have slightly different merging algorithms. However, all the tools essentially provide some way of comparing the differences between the modules visually.

[19] The reference at the end of each quote is a unique identifier. The first number was the number of the interview participant. The second numbers refer to the lines in my transcription. The quote contains verbatim conversation from the taped interviews. Although I did many informal interviews that were not taped, I only use taped notes in quote form. Notes in square brackets represent references to artifacts and processes that might identify the organization, so I have replaced them with more generic terms to maintain confidentiality.

> project are working on a different functionality in the same module, therefore the modules doing too much. (8: 1492-1495)

Explanations such as these reveal two important issues in the context of parallel development. First, the developers at Tool Corp. believe that better problem decomposition would resolve some of these parallel development issues. Second, these explanations of why parallel development is bad emphasize the coupling between code and people. People work on sections of code, and even more become associated with that code, experts with that particularly system functionality. When parallel development happens, two developers with different systems expertise, have to modify the same module at the same time.

If developers have a choice between work assignments then they often use the configuration management tool to find out whether a particular task requires generating a parallel version of the code. They use the evolution view provided by the tool to find out whether someone else is working on the module. The evolution view shows the history of an artifact's development at different points in time. Each time the artifact increments a version, then the tool records: the final state of the artifact (working, in-progress, unit tested, system tested or released as part of a public released system), the person who worked on that version and the version number. Over time, the evolution view shows the life of a module, from inception through different releases of the system, to the current state of development for a particular release.

> I'll look and see and if someone has it checked out, the module I want to modify and mine's not too difficult. I did this last night, I sent them mail and asked can you do this for me in your version (12: 2029-2031)

Developers use the evolution view of a module to find out whether anyone else is currently working on the code they need to alter. All the developers working on this project can use the evolution view. This allows them to make decisions about whether they want to engage in parallel development. Often if developers see that someone has the latest version checked out, they either ask the person working on it to incorporate their changes into that version, or try to work on some other task.

However, sometimes the developers can not avoid parallel development. Their changes may be too complex to ask another person to work on, or they may be too critical to postpone until parallel development can be avoided, so the developers check out another version of the module. At this point, even if they have looked at the view, the system flags them with a message telling them that they have made a parallel version.

When the developers have completed their changes they usually have to merge their code with the changes made by the other person.[20] The person who finished last takes responsibility for merging their work with the other person's. The tool supports merging by providing a facility that compares the two files and displays the lines that differ. The developer responsible for merging selects the lines that need to appear in the integrated module.

[20] Sometimes it is not necessary to merge modules at all, for example if the changes are hardware platform specific.

Merging can be easy when the developers have changed different parts of the module, for example if someone has changed the comments and another person has altered the functionality. Developers find cases such as these easy because the changes involve distinct parts of the module and that show up clearly in the merge display. In these easy cases the developer simply merges the modules without consulting anyone.

However, developers sometimes find that merging does not go smoothly. As the following quotes illustrate, while the developers referred to a single activity of merging, they had developed complex understandings of the difficult kinds of merging possibilities:

> So you can tell just by looking at the syntax, which is yours and which is theirs, and include all of your changes and all of their changes, and usually that's good enough. Err, sometimes when you both change the same lines of code, your changes don't include their changes, and their changes don't include your changes, is harder. (2: 310-313)

> What has to happen is the last guy who checks something in has to merge these two together, and merging to be honest is generally pretty easy, as long as the people aren't working on the same checks in the code. If I'm working at the top of the file and somebody else is working on something and the bottom of the file then it's fairly easy to merge unless those changes change the overall algorithm, then it gets messy. (3: 471-475)

> A lot of times, sometimes they'll make changes which are a little bit incompatible, and it's a lot harder to merge. Or sometimes they'll not even realize that they are affecting someone else's development and just go on ahead and not really clean up or take care of it. (5: 989-991)

The complexity of merging increases when the developers have simultaneously altered the same lines of code or algorithm to address different problems. When this happens the complexity of merging rises because suddenly differences become embedded in the context of how a module works, what problems and enhancements the developers were working on, and which solution developers chose to implement. It also becomes embedded in an understanding of the other developers' system expertise.

At this point the developer responsible for merging finds the other person who also modified the module.

> When that happens I usually get together with the other person and they're looking over my shoulder and we do it together. (2: 313-314)

> So, basically that person will get together with other people and the other person will oversee the merge. (4: 794-795)

They discuss what they did, explaining their programming strategies, the problems they solved, and the functionality that they believe the module possesses. They work together to develop a

shared understanding of both modules, and determine the functionality of the merged module. This activity often takes place as a joint merging effort. The developers sit around one terminal and select the lines that should go into the final merged module.

Parallel development involves multiple developers working on the same module at the same time. The developers depend on the same module for the work that they need to get done, and they end up depending on each other. Merging is the resolution of the technical dependencies that exist between the versions of the module. It also involves managing the social dependency among the developers working on that module.

Change Dependencies

Parallel development involves two or more developers making different changes to the same module at the same time. Software development is about changing code in one of two ways: either fixing a problem or adding an enhancement. So, in a sense all the dependencies that developers manage in their daily routines are change dependencies. However, I wanted to use the term more specifically to capture one kind of dependency that occurs because one logical change — one problem to solve, one enhancement to make — often involves multiple activities.

Developers know that changes to the product usually involve multiple alterations throughout the system. This immediately creates a change dependency among the pieces of code, and documentation, and test suites involved. Change dependencies create a relationship between all the pieces of the software that require alteration, because all the amendments must be made and integrated back into the product simultaneously. Failure to do so usually results in the system crashing, at Tool Corp. that means that the nightly build fails as the following developers describe:

> This is broken [referring to the automate tool designed to help ensure that all changes are checked-in at the same time], so sometimes you'll have maybe five or more pieces you have to [check-in] by hand, and there's room for error, and so sometimes a complete fix won't get [in] and you'll have a semantic error dealing with this say this type was defined in this body, but the type def didn't get [checked-in]. Then the nightly builds fail. (4: 894-896)
>
> I don't know that we really had a good solution with them, but we've been experimenting with change sets. That's a matter of grouping a couple of different changes together. Like say I change Module A and Module B, it's a way of saying these two things are related and you can't use this change without this change. (5: 995-998)

The relationship between pieces of code, is a technical change dependency.

However, change dependencies have social implications for the developers working with them. Sometimes developers have the luxury of making all the necessary changes themselves:

> There are certain times when in order for you to make a fix it spans several different [software components]. OK, and I nearly work in one set of code in GUI code but many times I'll make a fix in the GUI code but it also requires a corresponding fix in lets say the [language] interpreter, or the engine, and since I'm the one making the fix I go ahead and make the fix in both areas. (15: 2820-2823)

More typically one logical change requires several developers and other personnel on the project to make changes on various parts of the system. As soon as this happens, they become involved in the change dependency, dependent on the other developers to make their changes. In the following quote a developer refers to three separate activities, two of which must be carried out in parallel and must also be coordinated, so that the product remains synchronized with the documentation:

> For example this is a problem in a piece of code, and that has three tasks assigned to it, fix the code, my task, rewrite the documentation associated with that, [document writer]'s task and test it. (3: 719-720)

Change dependencies create links between developers working on the same change at the same time. The developers depend on each other to fix their code, or documentation, or test-suites, so that the final outcome works together and hasn't broken the system.

Demands for changes to the system do not appear from thin air. Instead the Quality Assurance (QA) group, consisting of testing personnel and management, decides whether the change needs implementing, the importance of the change and who will work on it (which involves finding out who has experience with that sub-system and who has time to make the necessary changes). Change dependencies are relationships among code and at the same time relationships among the people working on those changes. Further, developers depend on the QA group who ends up selecting the participants involved in this change.

At Tool Corp. the developers use the tool and some managerial techniques to manage their change dependencies. Usually one developer takes responsibility for the overall logical change:

> So a problem assigned to you doesn't mean that you are the only person who's going to resolve the problem. ... Now, say I was originally assigned to the problem then I have to coordinate when all the tasks have been completed I need to say resolve the problem. (7: 1356-1360)

That developer must ensure that all the developers make their changes and they get integrated into the new version of the system when they all work so that the product never has half a change in it.

The tool also plays a critical role in supporting change dependency management as a number of developers explained to me. One said,

> So, I don't really ever have to walk down the hall and talk to somebody unless I try to read the code and can't figure out what they were doing. I could look at my [evolution] view and see what, when they derived their change and I could probably tell from the comments why they derived their change and I can look at the specific change they're making even though it's still in a working state and they don't have it ready to give to anyone else yet. I can look at it as an observer. In addition to that, since when we create new versions we usually associate them with a task I can look to see what problem they are trying to solve, what enhancement they are trying to do, so I guess we have a lot of information about what they are doing. (5: 1008-1014)

The tool provides important contextual information that the developers would otherwise have to gather by asking people. The developer does mention an important part of the tool's role in supporting these change dependencies. Changes enter the tool as problems, a record in the problem database, describing the change and all the work that needs to be done on it. The QA group sub-divides this work between various developers, assigning each one of them a task, to fix a piece of code, amend the documentation, alter the testing scaffold so that the change can be verified as working, and so forth. The tool maintains all of this information, inside the central repository, as a series of hypertext-like links.

As the developer quoted above begins to realize during the description, the tool actually keeps considerable information about the current state of development for the project team. Anyone working on the code kept inside that database can see the state of all the other code, the problems being resolved, the current state of the work-in-progress, and as well as what happened previously. All this information helps the developers manage their change dependencies. It saves them having to engage in extended conversations, either in person or by electronic mail, trying to find out who's working on what fixes, and what state the change is in.

Expertise Dependencies

I define expertise dependencies as relationships between developers based upon their knowledge of a section of the product. Expertise dependencies arose in two ways at Tool Corp. First, a developer would find themselves assigned to work on a part of the product that was unfamiliar to them. Second, a developer works on a section of their code that calls a routine in another part of the product that they don't know. Both situations crop up routinely in software development at Tool Corp.

Modifying a piece of code that the developer does not understand very well carries some risks. Software modules interact with each other in ways that to a novice remain obscured from the source code. Modifying that code, without understanding those relationships, can easily break the system in unexpected ways and are difficult to understand. The developers at Tool Corp. realize this, and understand the significance of being the person that broke the nightly build.[21]

[21] Developers at a variety of companies spoke to me of a sense of embarrassment in being the person who broke the build. Cusumano and Selby (1995) describe the shame that developers face at Microsoft when they break the

So, they depend on people with expertise in that section of the code to help them work out what the code does, and how it relates to other pieces of the system:

> If I work on other pieces, like when I work on the baseline, because I know that this is more [developerA and developerB]'s domain, a lot of times before I make a change I'll go and do not really a code review but a design review or a fix review with them to make sure I because I don't feel like I understand this module as clearly as I do my own, I want to make sure that what I do here isn't going to break. (3: 558-561)

At Tool Corp. all of the developers, except for the newest, were considered experts with some aspect of the system. While the older developers may have skills that span several sub-systems, newer developers would know about the aspects of the system that they had worked on since arriving in the organization. All developers at Tool Corp. could easily identify the other experts:

> I would hope that in the [product] team each person would know what functional area each member of the team is working on. Put it this way, I would be really surprised if anyone doesn't know. ... I could do that, and I'm convinced everyone of them could do that. A new person has to learn that. (13: 2300-2302)

The developers also refer to the information stored within the tool for information about who that expert might be:

> You can also look, if you're in a particular module and you're confined within that module there's a created by for the last person who edited it. So, if I'm confused about an algorithm in a particular .ac and it's created by [DeveloperF] then I know to go up and say explain this to me. Yeah, the last person who modified the module may not know about the algorithm but they can usually point me to the person who does. (11: 1872-1876)

In this case the developer had not been with organization for long enough to learn who was an expert in that particular area. Instead he relied on the tool to reveal the name of the last person to modify the code. He would then ask that developer whether they were the expert or find out from them who was. Developers depend on the expertise of others to help them manage the technical dependencies that exist between code modules.

Finally, expertise also provides a way for the QA group to assign any changes that need making. The QA people use information about peoples' areas of knowledge combined with their current work loads to make decisions about who fixes a problem. This reinforces the expertise of the developers as well as ensuring that the problems get fixed as quickly as possible. However, recently the manager had begun to move people to different sections of the system as he did not want to have to rely on any developer to get a problem resolved.

---

nightly build. In some groups they must wear silly hats for the next day, and in others they become responsible for organizing the build until someone else breaks it.

Integration Dependencies

Although development takes place at the component level, software developers make changes to code modules, technical writers write sections of documentation, testers work on test suites that are sections of the testing scaffold, the system has to be pulled together as a whole. The nightly builds at Tool Corp. provide this function. Each night the tool gathers the latest changes to the code and compiles them, and then builds the whole system from its constituent parts.

One developer, known as the release or build manager, takes responsibility for ensuring that the tool builds the system, and puts the built version into a series of files that the developers can see and use. When the build fails, an error in one of the software components occurs and the build stops, the build manager tracks down the code that caused the problem and either sends or speaks to the developer that broke the build. As the current build manager explained to me,

> Oh OK, basically it's a matter of pulling everything together, the exact software product we're going to ship and putting it in an area so it can be tested and making it available for people so they can run on it and tidy things up on an everyday basis and it also results in, the build manager usually ends up tracking down integration problems like if two people have made changes which don't quite work together, I'm the first person who's going to see that probably. I look for that when the nightly build fails. (5: 953-957)

It was this comment that first led me to thinking about integration dependencies. For her, integration problems occurred when two different logical changes embedded in the code interacted in problematic ways. This makes integration problems different from change dependencies, where the fixes in one change must be managed by multiple developers. Integration dependencies operate at a higher level of abstraction, between all the development work happening on the system at a given time.

The build manager takes responsibility for ensuring that these integration dependencies become understood by the developers. The approach to managing them happens as conflict resolution, they may not be known until something happens that breaks the system, and then the build manager discovers the technical aspect of the dependency, informs the developers responsible, who then must manage the social aspect of that dependency.

The tool makes this kind of conflict resolution managerial approach possible because it reduces many of the problems associated with integration dependencies smoothly. It gathers the latest changes of the code from each of the developers, without having to ask them or trouble them to look through their directories looking for the newest working versions of the code. While automation is often characterized as deskilling work, in this particular instance, it removes an otherwise complex coordination process, as this developer explained to me:

> It removes the manual process of the developer saying I've finished with this, you can use this now. That's a pretty big advantage, I was a build manager for a part of it, [project name], for a couple of months. (2:406-407)

The tool will only gather files that have been checked-in and therefore tested, so checked-out working code, does not get into the nightly build. The tool helps developers cope with these kinds of integration dependency.

As well as supporting the gathering up of components, the system allows all the developers to see and work with the latest stable changes:

> Even in your own personal [sub-systems] you can see what the state of the parts of the project you are working on are because you get everyone's latest versions that others have checked in. When you reconfigure your [system] you see what versions you get, the dates on them, who owned them, who [checked them in], what changes they include. (2:408-411)

> Sometimes I can tell from just reconfiguring my stuff and I can look and see what, who owns all the versions that I just got in. I can see that certain things have been changing. (12: 2055-2057)

This allows developers to handle some integration dependencies. When developers begin a new assignment, they typically get the latest versions of all the code, by "reconfiguring" the system. This gives them a very up-to-date base point to work from as they explained:

> If I'm about to modify a section of the project, I make sure it's a current snapshot of the project of what my co-workers consider valid work and then do the modification from there. (8: 1534-1536)

> I guess one thing would be just the fact that if I come in the morning and I reconfigure [the system] I get the latest of everything and I generally don't even have to worry about it. I just know it's going to be there and it's going to work fine. Then I can just go about my business, having gotten everyone else's changes automatically. (5: 1021-1024)

Without the tool, finding a latest base point proves very difficult, and developers can be very susceptible to many changes that they have not received from other developers working on sections of the system that are remote to them.

It also gives the developers a way to realign their work as they get towards the end of the assignment. A common practice among developers was to check-out a piece of code, make the necessary changes, and then reconfigure their system to get the latest changes. Before finally checking the code into the system, they would re-test that the code worked with these latest changes. The tool gives the developers numerous opportunities to check whether their code works with the other system components, and because they use that, it pushes out the management of integration dependencies from the build manager to all the developers.

Historical Dependencies: An Organizational Memory of Action

Until now, I have only described dependencies that connect developers in the present. However, developers nearly always rework existing code, modifying it to fix repairs and add new functionality (Lubars, Potts, and Richter, 1993). When developers reuse old code they often find themselves trying to work with code that someone else wrote. The job of development then becomes the task of aligning your efforts with the work of the previous developer. The complexity of working with other's code increases when the developer who originally wrote the code has left the organization or is assigned to a different project (Fischer and others, 1992). Often developers rely on decisions taken and changes made in the past; I call these historical dependencies.

At Tool Corp. the tool helps the developers manage their historical dependencies and provides an opportunity for developers to work in the past readily. Without the tool, and I saw this in other organizations, developers refer to previous versions of the source code, but do not use other information that the tool at Tool Corp. provides. I have already said that the tool relates changes, "problems," to specific changes in the code, through hypertext links. As the manager explained, unlike other systems, such as Lotus Notes, which also has an archiving facility, the organizational memory at Tool Corp. links the changes made with the actual code, the "data" contained in the system. He compares a previous development project that he led,

> All design was done through [Lotus] Notes, we didn't do fancy documents, and the great part was we had a history, a chain of events, of why things happened. So that was more of a project history, but what's different about Tool Corp. experiment really is instead of attaching it to a string, or a question or a topic, that data is attached to the data. That information is attached to the data. It's attached to the subsystem itself rather than the topic, and that's not better or worse as much, that wasn't by design that was more a side effect of, we have a CM system, our repository's data centered so we attach it to the data. (13: 2214-2220)

The tool manages this using its problem reporting facility. When developers alter any artifact the tool forces them to link the new version of the artifact to one of the changes in the problem reporting facility. The CM tool stores these links, and over time they build into a memory of which artifacts changed as a result of a certain problem or enhancement. In this organization the memory has been growing for over 2 years. These links are augmented by a free form comment field where developers can describe their changes. The CM tool stores the comments so the organizational memory contains problems and enhancements, the artifacts changed, and often descriptions by developers of how they implemented the solutions.

All of the developers use the organizational memory to go back into the past and learn about the work of the predecessors. The different things that they look for emphasize the variation between different types of historical dependencies that exist. A number of developers described their use of the organizational memory to go back to find a single module.

> That's I think the main advantage for me is that I can look at a module and say OK who wrote that last and then if I have to work on that I can go back and say OK so and so worked on this now what did you do here things like that. (17: 2995-2997)

> All the time, it is really useful. If you're making a change in some code it can be hard to remember what someone else did before. You can look at the comments, that tells you who made changes. You can see the tasks which were assigned to the code, and the problems which they were trying to solve. That way you can even see who was working on it. (5: 2435-2438)

As these quotes illustrate, developers at Tool Corp. go back to a single module to try to learn about why certain coding decisions were made, or what problems the developers were trying to solve, and how they implemented the changes. The developers search for a context for their own work on the module. The current developers find that the process of upgrading existing code does not only involve making the technical changes, but also learning about the social context in which that module evolved. Historical dependencies start with relationships between current and previous versions of systems components; however, they also create relationships among the developers who worked on those versions.

Developers come and go at Tool Corp. as they do in all software development organizations. The tool provides a way for developers who have never met each other, who will probably never work together, separated in time, to coordinate their work. As the manager explained,

> There is an activity going on today between [DeveloperC and DeveloperD] that they are out there making a fundamental change to the way one of architectural mechanisms. They are working in a piece of the system nobody has an in depth knowledge of here, the original developers for that were gone over a year ago. So they are going, the only source for what's and why's of that is through the system, thankfully there is this big database very big database that's got this history, So they're out there looking at the history and basically we assemble the history of the why's. (13: 2192-2197)

I followed up with the developers themselves who were indeed using the tool to build that context about the evolution of the architectural mechanism.

The person in charge of interface development uses the organizational memory for similar reasons. In his case some of the original developers have left the company. Interface development was also distributed across many developers until the management saw a perceived need to try to unify the interface to the entire product.

> I do that all the time with GUI code because I didn't actually write the code, I maintain it, so then I sometimes have to wonder why it's done this way and what happened, you know there'll be twenty versions of I'll go back through all twenty versions, figure out when did this actual piece of code get in, entered, and see what the comments were. (15: 2705-2708)

Because he often found himself editing parts of the interface code that he did not write himself, he consulted the organizational memory to see what the developer who wrote the code had said about the task at hand. This provided information about which kind of solution to pick.

Even if the original developers existed, sometimes the original artifacts for a system get misplaced within an organization. The organizational memory keeps them all together within the system itself. This saves everyone some time as the manager explained,

> I had to go and figure something out just the other day that was, well it was related to one of the original design documents for the system. I went into the database, I searched for this thing, you find this thing, now how did I use to do that. I used to do that, I'd probably go to a person, and if I could find that person, but now I'm going to, in effect I use this term favorably, a repository which physically and figuratively is a repository. It's got not only the documents, but the history of the documents, the reasons the documents changed. (13: 2202-2207)

I have described the organizational memory as a part of the system, the portion of the tool that links problems to artifacts within the system. Because the tool serves as a repository to store all the components, the whole system acts as a memory of specific points in the evolution of the system. Tool Corp. puts all the documentation and test suites into the system, so technical writers and testers, can also go into the tool and rediscover the missing context that provides a needed explanation. The senior tester revealed another important use of the organizational memory in his work:

> I'll see something that's really strange, and I'll think to myself, this is really strange, I could talk to a developer but sometimes because there's so many interdependent pieces a developer couldn't really tell me if what I saw was really expected, but I can go back to a log file, like a year ago, and look up the run, and see the complete run. I can go through and compare, and see if its that dramatically different. Try to understand what sequence events led up to a certain happening which I didn't understand. (14: 2596-2600)

In this case the context that he sought, not a development context, but a testing one, could not be put together by any single developer. In this case, the tester clearly felt that he was as reliant on the tool as the developers working with the echoes of those who had left the company. In this case though, he was dependent on a context that spanned the expertise of the developers, and that they might not be able to piece together for him. The fact that the developers find it difficult to understand the system as whole has more serious ramifications that I discuss in the section on group-level dependencies.

The tool also had special uses. About two thirds of the way through the study the company decided to change a naming convention used throughout the system. The name was embedded in the product, appearing on screens and named in commands. The manager assigned a number of developers the task of going through the system and changing all instances of the old name to the new name. Fortunately for the developers this name changing had already happened once before, and the code changes were linked to one problem describing the previous name change.

They began with the problem and found all the artifacts that changed: those containing the name. This found most of the instances of the name, only excluding modules created after the last name

change. The developers also used the free form comments to find out whether the previous developers had experienced any difficulties when they did the earlier name change. As one of the developers responsible for the name change describes, this saved the developers a lot time:

> We looked at the tasks and the problem, there was one for each database, from the last name change. That gave us the list of things which would probably need changing. It was the basic set of changes. We actually needed to do less, because last time we were smart, and used defines instead of having things everywhere. Last time we did the name change we had to grep on all the files, however, some information containing names was stored in the database, and we couldn't grep on that so we missed it. The next day when we tried to compile we discovered that the system didn't work because we'd missed making the corresponding changes in the database code. We couldn't grep on that code which is why we missed it. (5: 2425-2431)

nstead having to search the code manually using UNIX tools like grep, they had a head start. As everything gets stored in the repository they can search that, and then can use the organizational memory to organize that search.

However, the organizational memory does not operate without creating its own problems. It is worth examining the problems that arise when developers try to use the memory sometimes. These problems often arise because the fill-in fields do not contain useful information, as one developer noted:

> Sometimes, it's helpful, sometimes it doesn't. It all depends on the input of the people. (15: 2708-2709)

Another developer offered an explanation for why this might happen. She said,

> It's a minor point. I don't fill out all the fields as well as I should, they take a long time to fill when you're in a hurry. It's annoying to me when I try to find out more about someone else's work that I try to use the tool and read these fields and they're empty or meaningless. So I know it can be upsetting. (6: 1248-1251)

When development proceeds at a relaxed pace then people usually take the time to explain what they did in the comment field. The pressure of tight project deadlines encourages developers to write less in the comment field. When other developers review these comments they do not understand what happened in detail that makes the comments almost meaningless.

When developers do not find the comments useful then they must find other ways to compensate for this missing data. One developer summed up his own strategy,

> Well, for one thing when you get assigned a task, you know that task is assigned to you, but you still go to [problem reporting system] to find out exactly what this task is about. They only give you a test synopsis, it doesn't really have a description of what the problem is, how you go about solving it, so you go to

> [problem reporting system] and find out. All lot of times, people enter problems or assign problems they don't really put in good description about how this problem should be fixed or why we need to fix it. So that's when you need e-mail support for further discussion of this problem. (7: 1374-1378)

When developers can not get all the information that they need they rely on other methods of finding out about what happened. They can rely on other developers' expertise of sub-systems, or the QA group's knowledge of what change the task was a response too. I have described these as other kinds of relationships, expertise and change dependencies, and they are. Software developers must manage a cadre of dependencies simultaneously if they are to build any working systems at all.

A rarer problem that Tool Corp. faces is that some of the configuration management tool was built before the problem reporting facility was added. When developers make changes to these parts of the system, they have no information from the organizational memory because it did not exist at that time. Fortunately this does not happen often.

The organizational memory provides new opportunities for the developers to manage the historical dependencies that arise when they make changes to code. Sometimes the tool does not provide the developers with the kinds of information that they have become accustomed to though. These disappointments illustrate the increasing tool dependencies that the developers have,

Configuration Management Tool and Practice Dependencies

The tool offers the developers some support in managing the dependencies as I have described above. However, their sense of configuration management, and it's importance, as well as the instantiation of that in the tool itself, does cause them to become dependent on the tool and their practices. I have already described cases where the developers do rely on the tool, and in this section I will detail two other important ways that the tool and practices of configuration management shape the way that developers understand the development process at Tool Corp.

One developer explained, he relied on the tool implicitly,

> I think it manages the process, a lot of things that you normally do on the fly remembering in your head, it manages. Mainly versions of files, that's the main advantage CM gives you. Sometimes you put a change in, and you go in the wrong direction, and you want to go back, and it's nice to have that officially somewhere. (2:396-398)

Backtracking is a strategy that most developers follow whether or not they have a tool storing the code. Even without the tool developers working on code keep different versions of the files so that if their latest revisions do not work, they have somewhere to go back to and start again from. However, because the tool managed the code the developers did not keep versions of the code.

However at Tool Corp. the developers depend on the tool to organize their work using the problem reporting facility that I described earlier.

> I like to use the tool to organize my work. I use the [problem reporting] facility. I create tasks for just about everything I do. For a while I even created them for work which didn't necessarily have things in the database which I would have to check out and change. (6: 1198-1200)

> So it keeps track of all the problems which I have assigned to me and I can put priorities on them, so I know which ones I'm going to do first, also it has a field for estimated duration, so I can get an idea how long it will take me to do everything, and I can budget my time. (2: 326-328)

This works well because the developers control the problem reporting system:

> Well, we have complete control actually because we are all have the [problem reporting] admin privilege role. Well, whenever I find a problem in my code, or other peoples, but particularly in my code, I make a problem and submit it and I automatically assign it to myself. So that's the first way. Then we can go back in and modify the priorities and length of time so we actually have full control. (6: 1219-1222)

The control over the problem reporting system was not intentional in the beginning. As I previously explained, the tool relates problems to changes in the actual code itself. In fact the system does not let developers work on a piece of code without having an associated change in the problem reporting facility. This used to prove problematic, as developers ran out of problems before the QA group could meet to generate new ones for them. The project manager decided to let the developers all have this privileged role where they could create and assign themselves problems, so that they could get on with their work. Along with creation and assignment of problems, the developers can now access and revise their time estimates for fixing problems. Other developers can also look at the problems assigned to a developer and use their time estimates to align their work. For example, if they believe that someone else will soon finish their code changes, they can elect not to make a parallel version of the code, and wait instead.

However, the tool only provides a starting point for developers to organize their work. One developer summarized how he depended on the problem reporting facility to arrange his own work load, but also observed that the initial display of the problems did not convey all the information necessary:

> It's nice, you come in the morning and get a mail message, these are all the things the problems assigned to you, just look at all of them. No-one actually has to come to my office and say this is a bug, it has to be worked on, I just know because it's automatically generated and sent to me. So I look at that to figure out all the things I have to do. Obviously I got to, the listing isn't complete, its kind of a synopsis of the problem so then I have to go back into our tool and look it up

> and read the description of the problem and gage whether or not I want to do this or not. (15: 2691-2696)

The developer used the tool as a starting point to learn about the problem. As we have seen in earlier examples of the developers work, they often leave the tool, and search for other developers in order to conduct their work. However, the tool plays a useful role up front giving them information about what's expected and pointers to other data sources. Sometimes the developers reliance on the tool ends up hurting them though:

> In fact I had to, a whole bunch of problems got moved from someone else to me, but they never got updated in the database, and so she thought she was working on them and I thought I was working on them too. So I ended up having to go and mark them all mine in the database. (12: 2045-2048)

In this case the developer who had the problems originally assigned to him, and the new developer had both started fixing the same problem. Now that the tool holds information about work assignments the management group must continually align their decisions with the information presented by the tool. When the two, the tool and the verbal decisions, conflict then problems such as these occur.

Another developer described a different kind of tool dependency to me, but an important one:

> If you have a lab partner they might have set these things up in a completely different way. In [the tool] the system sets up everything in a standard way. It's easy to find out what is going on. There's rhyme and reason to it all. I can understand through graphs and views, in regular development it's looking at makefiles. Talking to someone is one better, but you don't have access to the person, you can sort of understand. (3:729-734)

This developer knows how the layout of the software artifacts provided by the tool, as all the developers at Tool Corp. do. They rely on the standard presentation of information to make decisions about the way that the code works. They use this to learn about code that they must change, and its relations to other parts of the system.

Interface Dependencies

The interface developer must also manage a set of dependencies unique to the interface of the system. As he explains,

> Most of the time, especially with the stuff I work on, the interface, people will log a lot of bugs against the interface because that's what they see. So they see the interface crashing, so obviously they think it's in the interface, when in fact it really turns out to be that other piece of code somewhere else is not working correctly. (15: 2743-2745)

The interface depends on the functions of the code that sits beneath it. The interface often simply reflects the underlying behaviors of other sub-systems of the tool, for example, presenting the database of code, documents and test-suites to the user, or showing the problems in the problem reporting facility. However, when other developers or customers run across problems either they assume that the interface has generated the error, or they do not have the time to explore other possibilities, and so problems get assigned to the interface developer that do not belong to his section of the code.

All of the developers sometimes get problems that do not belong to them. However, the interface developer gets randomly assigned problems more routinely than the others. He copes with these dependencies by spending considerable time tracking down the sources of the errors. It often leads him into sections of the system that he does not understand, and then he relies on the tool to help him understand the pieces of the system, or help him locate the experts in that area.

4.3 Group Level Dependencies

In the previous sections I concentrated on the dependencies that individual developers manage in the course of their work. These dependencies require developers to work together, to engage in forms of articulation work, coordinating their efforts. Group level dependencies differ in one of two ways. First, they involve interactions among groups of people. Second, these dependencies may act within one group, but involve the entire team either acting as a whole, or sharing a common understanding.

How those dependencies manifest themselves depends on the organization. In a small development environment individual developers manage these dependencies. In a larger development organization these inter-group dependencies may be managed by formal hierarchies, or special groups of individuals, as I shall show in the chapter about Computer Corp.

Life Cycle Dependencies

Tool Corp. engages in a number of forms of parallel development. I have already described the case where two developers work together on the same module. At the group level two teams of developers often work on different platform releases at the same time. This was the case at Tool Corp.

> With the [product] release coming out the code on the [platformA] side is very chaotic it's changing every day. They do re-integrations every night, re-compilations, and I didn't want to thrust a very beginning porting development into that environment. So, I got the latest version, the pre-release was the version I chose and I ... pulled it, I wrote a script to pull all the source code out and laid it out in a more traditional format. Then I copied that to the [platformB] and began my development effort. ... If I decided to stay within their paradigm then they

> would see [platformB] defs. Also changes which I have made, which I didn't want to impose during a chaotic development cycle. (11: 1829-1840)

This developer was responsible for the initial developments on the new hardware platform, and was soon joined others to work to release a product. Although for now they have managed to isolate their work from the other platforms, they must eventually re-integrate it back into the main development database, so that the tool can organize the code as it does all of Tool Corp.'s code.

At the time the developer decided to isolate his code because the pace of development on the two different platforms was very different. Platform A code was approaching the end of the development life cycle. Feature freeze, the point where no new features can be added, had occurred. The system was undergoing extensive testing and modification only. Platform B was in the initial development phases, where the developers did not necessarily want to even conduct a nightly build, but work on big sections of the code, rather than tweaking it. On both sides the pace of the cycles would have proved disruptive to the development efforts of the other. The hope was that once Platform B, the new platform, was as well developed as the older platform the two would share one common life cycle managed by the tool.

This happens even on the same platform. When two developers made a significant change to the product, they wanted to isolate their work from the rest of the project team so that they could develop and test at their own pace before merging back into the faster pace main project development. As one of the developers explained,

> Well, [DeveloperG] and I were working on, we were doing a larger project but we had quite a bit of overlap in our work. So we were able to use the tool to isolate our work from the rest of the group and share our work with each other. Well, just that it allowed us to isolate the work from everybody else's. We were able to create parallel branches from what everyone else was working on, and putting special tags on what we created, the rest of the people didn't see, didn't get it bound in. It kept them from using it in their product... When we were ready to merge it in, we had to go back and merge it. But it maintained the history and the relationships. (6: 1189-1194)

In both of these cases the tool helped the developers to reduce their dependencies on the pace of development. It worked both ways too. The developers working on testing and fixing the main development did not have wait for these two developers to finish these complex changes. The others could test, fix, and build the system, without getting extra platform or project code. Those working on the project and the new platform did not have to structure their work so that they could test, fix, and build their systems more quickly.

"Big Picture" Dependencies

While the tool provides support for many of the individual dependencies that I have described, it did not provide any mechanisms to help the developers visualize the system as a whole. The fact

that this "big picture" was missing and that the developers could not easily put it together emerged in a number of the interviews. It also surfaced in their actions.

When development started on the configuration management tool, Tool Corp. assigned a few developers to the task. At this time the development process was managed by a group of friends, who talked over the walls of their cubicles and in so doing stayed in touch with all the aspects of development, as one of the original developers explained to me:

> In our last building, we were in this tiny little office, that really had no separate offices, and the cubicles were about this high [she motions about 4 foot] so you could see everyone in the whole building and all the developers there were only about six of us at the time, in this tiny little area which was just these short cubicles, and we talked to each other over the walls all the time and things like that. When we moved into offices when we moved into this building it so hard it was really just people in offices along the wall, we were the only people who were developers at the time so it was just really hard for us to get use to it, basically there [in the old office] you could keep up to date. People would have conversations well how should I implement this you know over the wall and everyone got to hear it. (5: 1026-1033)

As the product became successful, the company grew and the development group also got larger. The developers began to specialize in their own sub-systems and lost their sense of the whole system at the same time. While the developers understood their own sub-systems well, and many had worked on different sub-systems, the developers did not know the conceptual structure of the product being developed. This problem was exacerbated by the changes in design that occurred throughout its development that altered the system. One developer, who had not been with the company long described the isolation, and the affects of that as:

> there's a lot of different sections, which call routines in other sections, that's sort of, you pretty much need to know the big picture of the group, as well as the individual part. I have routines which make calls, type creation calls go to some of my routines, hands me information I need. ... So you pretty much have to know what other people are doing, although its kind of scary because we are working on our own particular areas and no one really knew what it was going to look like when it all came together. So, we did have a lot of design meetings which was really good, that was really important in the beginning to get a good picture, but once we got specifications, we started into developing our little portions, we had an idea of the other things were, but not of how they were developed... (4: 920-928)

His concern was that no-one really understood how the system would look when it was all put together. Although the system pulled together code for the nightly builds the developers couldn't visualize that whole. This has important technical consequences. Because they could not see that whole, they had trouble recognizing how the parts would fit together, what those interactions would be. It also crippled their ability to practice software techniques such as re-use. Because

the developers did not know what their colleagues were working on, they couldn't make use of anything that anyone else built.

The developers had a number of strategies for trying to cope with the fact that they didn't really understand how the system would fit together. Some of the developers tried to use the tool to find out about the system:

> What is essential, in a group, is that people need to know what other people are working on, sometimes, that's the weakness we have here, sometimes people don't know what other people are working on, it [the tool] doesn't tell you that. ... I guess you could look at the problems the person has completed and read the descriptions and get an idea, but usually there's not enough time, that's pretty time consuming, usually, what you want is a one paragraph description which tells you what everyone's been doing. (2:413-419)

The problem reporting facility though, tracks the changes in the system at a very low level, the modifications to the code. The developers wanted a more generalized description of what their colleagues were doing. One developer drew up an architectural diagram of the system in his spare time. Architectures,

> permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems. (Garlan and Perry, 1994; 363)

The architectural diagram consisted of blocks and lines that represented conceptual units of the system, and the relationships between those units. Many of the developers had this architectural diagram pinned up on the walls of their offices. Many of them had annotated the original diagram by hand to show new and different connections between the sub-systems. Another developer referred to the importance of communication, especially as individuals specialized:

> I like fiefdoms of responsibility. It lets people work their own style and if, as long as they are responsible enough to realize that people are using this. To share that, separation of code I like because it gives a sense of ownership and responsibility, and pride when something works well for a whole lot of people after he spits it out. But to counter that isolation effect you need to be as communicative constantly about everything you do, where you can find out what other people are doing and if it's useful to you, and if anything you're doing is useful to anyone else. (11: 1914-1919)

However the developers had just about exhausted their communications networks. Most of the developers received in excess of two hundred messages daily, because they belonged on many different company mailing lists. They did use e-mail to initiate discussions about the system, but these discussions competed with all the other e-mail messages, and were often ignored by other developers.

Other developers turned to the documentation,

> Most people don't have a clue what anyone else has done, even at the highest levels. Like I know [DeveloperE] wrote Tool Corp. make stuff, but I have no idea how he wrote it or what it does. Besides from reading documentation about it. (12: 2078-2080)

However, during initial phases of the development the documentation often does not exist, or remains out of synchronization with the development effort. Also the focus of the documentation was on users, so developers got little information about the implementation of the features described in the documentation.

Even the manager recognized the problem,

> Once you have about ten people in a development organization you rapidly realize that you don't know what every body else is doing. You can't know what everybody else is doing, when people come from a big company like me are very comfortable with that, well I'll say that we're used to that, not knowing and you can deal with your day not knowing, realizing I don't know what everybody else is doing, people that haven't been in that scenario, and we have several are not used to that concept at all, they feel like there is a major cultural shift which has happened which they're right, not good. The fact that you don't know anymore what everyone's up to is not a plus. (13: 2232-2238)

He described it as a shift, and during my time there I saw the organization grow very rapidly. I did not find any empirical evidence to support his claim that developers from bigger organizations would feel more comfortable with not knowing what their colleagues were working on.

As a group the developers at Tool Corp. simultaneously depend on each other, and their managers, for a sense of the whole system. Furthermore, most of the developers had been at the organization when they had some limited sense that they did know what everyone else was working on. As the development group grew in size and specialized, they became more distant from other people, and other sub-systems and they lost their ability to put the product together as a group, share technical solutions across the boundaries between the different sub-systems. The tool, and forms of communication, that the developers had experimented with to date had not helped them re-find the whole, from the parts.

Testing Dependencies

The testing group works closely with the development group. During the time I spent at Tool Corp. this relationship grew steadily closer because the developers went from the middle of a development project towards final release. The closer that they got to final release the faster the cycles between testing, fixing, and building the system. In the last few weeks they tried to build

the system at least twice a day, and everyday at the weekends. The developers also expanded into testing roles.

However, I have elected to treat the testing group as an essentially separate group from the developers. They perform a different function from the developers. They are primarily responsible for comprehensive system testing, particularly integration tests, that check if different sub-systems of the product work together. However, because they work so closely with the developers they depend on the developers, and on the code that they write in their own work.

Testing dependencies arise because test suites, automated programs that run the system through a number of scenarios, need to be constantly updated to reflect changes in the product. The testers have responsibility for ensuring that their test suites accurately test the product's functionality, but as this evolves throughout development, so they must evolve their own checking programs.

> The reason why testing usually fails, automated testing technology usually fails, is because of the problems with parallel development. The test scaffolding which we have here is totally dependent on the product, there are pieces of the product used in the test. Some common technology, so in that sense its parallel development. We're going off of that technology, and in fact every, like last week there were two changes to pieces of the product in [New Version] that affected the test scaffolding. (14: 2479-2484)

The situation gets more complicated because Tool Corp. has a variety of stages of testing too. System testing needs to be very rigorous, and includes a period of internal use. The developers use a version of the system that has passed the initial unit tests of developers, and a series of test-suites written by the testers. While the testers continue to test for more obscure and unusual problems, the system also becomes used internally, as one of the testers explained:

> We're actually testing on any given day a patch to a particular release, for one customer, [newest build] which is our bleeding edge version, [older partially tested build] which is kind of beta version of the software we use [in-house], or the next release of the software. Now the interesting thing is that the expected behavior or the expected results for any of these is totally different, and not only that, the test scaffolding software because it's built on some of this technology is also dependent in, and is actually different for each of these. (14: 2577-2581)

In each case the system varies, and the test-suites also vary to compensate for those changes. The testers have a number of technical and social strategies for managing their dependencies with the developers. First, technically, they store all their test suites, the test scaffold, inside the configuration management tool. This helps them to keep track of which test scaffold belongs to which version of the tool.

At the same time they also need to know how to adapt their test-suites, so they need to know what the developers have changed. They do this in two ways. First, they use the problem reporting facility. This lets them see what changes have been recently completed. Second, they

attend the development group meetings routinely, and communicate with the developers frequently. The testers need to maintain high levels of contact with all of the developers so that they can determine exactly what the impact of the developers' changes will be on their test scaffold. In these social ways the testers manage their dependency on the developers.

4.4 Inter-organizational Level Dependencies

Inter-organizational level dependencies come from sources outside the software development organization itself. In much software engineering literature the focus of building software stops at the organizational level. The implicit assumption is that software development organizations act in a vacuum, entirely on their own, with no context. This has never been true. However, with the arrival of "open systems" — systems that have a high degree of compatibility with other systems available in the marketplace — this has become even less true. Tool Corp., like all software development organizations, needs to maintain strong connections with other organizations in their software development world. In this section I describe two external dependencies that influence Tool Corp., relationships with vendors and customers.

Vendor Dependencies

Tool Corp. also depends on the vendors of the products that it uses to run its technology on. During the development of the product, a new operating system was released by a large software development organization. The management at Tool Corp. realized that if the product did not run on that particular operating system they would miss out on a section of the market for configuration management tools, those organizations that would upgrade to the new operating system as it became available. Software development often happens in very technically heterogeneous environments. Developers use a variety of machines and operating systems, so that they can test their product out on all these different combinations. Tool Corp. is no exception and their development environment is heterogeneous. However, when an organization builds a tool that is designed to control the development process, then it must run on as many of those platforms as possible to achieve a high degree of penetration into other software development organizations.

At the point when the new operating system became available, the management of Tool Corp. had little choice but to begin developing a version of their product that would work for that technology. This contrasts with the traditional view of software development, where authors of professional and academic software engineering literature assume that organizations have absolute control over their development schedule. At Tool Corp. other vendors played an important role in determining when Tool Corp. began developing another version of their system.

As well as influencing when Tool Corp. begins development on a version, other vendors shape what exactly is development. If substrate technologies, like hardware platforms, and operating systems, do not contain certain functions, then Tool Corp. can not provide features that rely on those functions. All configuration management vendors wish to develop ways to support

geographically distributed developers. (Since the time of this study several companies claim to have resolved this particular problem.) However, to provide distribution, configuration management tool vendors like Tool Corp., depend on these substrate technologies to offer fast and reliable ways of passing information between databases. Some senior management felt that current commercially available technologies did not do a very good job of this.

Customer Dependencies

Tool Corp. and the potential and existing customers depend on each other. Tool Corp. depends on its customers somewhat to define the requirements for the product that they build. The customers depend on Tool Corp. to provide a product that they can use, and what is more important, one that continues to receive support and enhancements over time.

Requirements

Indirectly customers, and potential customers, influence the direction that the product takes by buying other systems. All "open" systems, like Tool Corp.'s product, need to support a variety of integrations between the core product and other systems. A configuration management tool should supply connections to other development tools, debuggers, testing tools, and software development environments. The market determines the systems that should have integrations as this developer explained:

> Well I'm involved in the integrations team. So that involves basically taking our tool and taking whatever other product tool that our customers want to use and integrating it with the product. (17: 2981-2982)

He was not referring to specific customers in this instance, but the pool of potential tool buyers out there. However, specific customers do influence some of the development, because they can submit problems and enhancements into the problem reporting facility. Although the customers have the option to send the problems electronically, often they call their concerns into the support group who then enter the problem into the problem reporting system.

> So once its entered it comes into, from a customer outside, in to the support group, They look at it and enter it into the correct database, then I'll look at the problem, I'll try to duplicate it, unless its obvious and then I'll put it in review. From in review it goes to either assign or deferred, if it's a problem that we decide we can not fix or decide that it doesn't make that much difference. Otherwise we'll assign it to the right developer. (10: 1723-1726)

Tool Corp. depends on its customers to help them shape the direction of their system. This happens indirectly, through the market, as well as directly, from change requests arriving to the developers from existing customers. Tool Corp. has further recognized this dependency, as have other organizations, by having a series of yearly "directions" meetings with their customers. At these meetings that typically last a few days, they invite customers to share their experiences of using the tool in their own development environments. They also share some of their ideas for future development to try to gain some initial feedback on the worthiness of the proposals.

Support

I attended one of these directions meetings and it helped to alert me to the fact that the customers also depended on Tool Corp. During the meeting that I attended it was clear that Tool Corp. executives wished to solicit ideas about what future directions would sell and increase their market share. At the same time the customers bought their own, more specific, agendas to the table, consisting of enhancements and changes that they would like for their own development environment problems.

Customers depend on the fact the future evolution of the product will, at least for a time, remain compatible with the versions that they have. Tool Corp. recognized this. During my time at the Tool Corp. the developers were simultaneously developing two versions of the newest product. New Product was a completely new product trying to capture a new section of the configuration management market, while New Version of Old Product was designed to provide existing customers with extended features.

Even within a single release, once the developers enter the final stages of development, some customers already have beta-versions for testing, they recognize that customers depend on what they do:

> Currently any enhancement must be scrutinized so we have meetings on that on any enhancement, at this point. Earlier on I had total control, but at this point, documentation has been written and the product is in customers' hands and we can't go adding all kinds of new features which documentation doesn't know about, marketing doesn't know about (8: 1571-1574)

Tool Corp. and its customers and potential customers depend on each other, for requirements, future directions, and support. The organization handles these dependencies in two ways: the marketing department analyses the state of the current marketplace and makes recommendations about possible future directions and the managers get together with customers and discuss their needs as well as revealing some of their plans for future versions of the tools. As these requirements become more specific the developers get to work on them, implementing them as changes to the existing system. At the same time the customers depend on Tool Corp. to make alterations and provide support for the product that they use.

4.5 Summary

In this chapter I have described three levels of dependency relationships, individual, group, and inter-organizational. Particularly I have spent some considerable time devoted to elaborating the kinds of dependencies that individuals manage at Tool Corp. This is a function of the time and access that I had to the developers. During my time at Tool Corp. I interacted daily with the developers, and like them, had to schedule specially arranged times to discuss my work with the managers of the organization.

I also described three vital components of the dependency relationships. I characterized both the technical and social components of the dependencies. I found that it was impossible to separate the technical elements from the social ones, the developers never did, and I believe that this fake separation contributes to the fact that dependencies remain ill-understood in academic circles. I also documented the methods that the developers used to manage these dependencies, to resolve them, at least temporarily, so that they could carry on developing software.

Dependency management forms a critical part of software development work at Tool Corp. Because the tool supports some of this vital coordination work, I did not discover the importance of managing these dependencies until I visited Computer Corporation that I discuss in the next chapter.

Chapter 5

Case 2: Large Computer Manufacturer Seeks Good CM System

> What you really have is um complex dependencies so it's a layered dependency problem so it's not like every dependency is exposed in the first order to every other space it might be second or third order dependencies out there. So in that sense it has this dimensionality to it that makes it feel parallel but it really isn't parallel it's just a dependency tree that's really um weird, really hard to even visualize what it might look like or even, we're thinking of starting an initiative that would be a whole task force or specially chartered group to examine dependencies. Just because you know that is such a hard problem for people because it bites, its enterprise wide dependencies, right so how do you manage them, well right now we don't manage it, we stumble over it, and try to solve it every dependency one at a time (5: 1388-1396)

Tool Corp. provided insights about software dependency management. The developers and managers find themselves in a web of these dependencies; to build software they must resolve them, at least temporarily. This chapter describes dependency management at Computer Corp. and extends the analysis of dependencies by providing supporting evidence for their existence and the forms that they take.

Also Computer Corp. illustrates some of the challenges of managing dependencies in a larger development effort. As has been observed, the complexity of development rises sharply, exponentially, when more people work on building the software. The study of Computer Corp. confirms this assertion and shows that the complexity of dependency management rises as well.

5.1 Welcome to Computer Corporation!

Computer Corporation is a large computer company that has its headquarters in Silicon Valley, California. Computer Corp. began as a spin-off from another large computer company in the 1970's. Initially they concentrated on building real-time systems that had a high degree of fault tolerance. They marketed hardware and software that processed data quickly even when parts of the system failed.

Today Computer Corp. is a Fortune 500 company. It has operations in over 40 countries and profits in excess of over US $2 billion annually. The company has enjoyed a period of financial growth in the nineties, with profits steadily increasing. Currently the company employs around 8,000 employees in over 150 locations. Computer Corp. sells its products and services to a

variety of industries including: telecommunication companies, manufacturers, health care and insurance industries, and banks. The products and services that they sell are high performance real-time systems. They emphasis the kinds of activities in these market sectors that require real-time data processing.

Software development has a different character at Computer Corp. than it does at Tool Corp. The company has many locations, and even though much of the software development takes place in Silicon Valley, as the company has grown the development has become geographically dispersed. The company has approximately 700 developers working on the main product suite, mainly located in Silicon Valley, but with some geographically remote development efforts.

Recently Computer Corp. has expanded its operations to take account of the changing nature of the marketplace in which it competes. Initially, the company they sold their own proprietary hardware and software. They still continue to build both hardware and software, but now can not expect potential clients to have computer systems of theirs in place. As a reaction to this phenomenon Computer Corp. has moved towards open systems and now provides integrations between its own hardware and software and other commercially available systems. Like Tool Corp., Computer Corp. sells its products on the open market; however unlike Tool Corp., they also offer customized versions for special customers who can afford them.

The impact of open systems has also changed the character of software development inside the organization. One obvious affect is that the developers have begun to use commercially available platforms for development instead of the proprietary ones. Suddenly the development environment has become more heterogeneous, and consequently requires new methods of management. One significant change was that the company needed to revise its configuration management strategy.

Configuration management used to be handled by an internally built system that worked for the local environment. The system was relatively sophisticated tracking the development of different versions of the product including customer-specific changes. However, priorities changed within the company and instead of maintaining the internally built system the company decided to buy a commercial tool. This has the advantage of then freeing Computer Corp. from the obligation of maintaining it and updating it, and hopefully brings the added advantage that the configuration management vendor will keep up with the latest advances in configuration management technology.

The configuration management group reviewed about 12 systems before deciding to buy Tool Corp.'s system. During my time there Computer Corp. had already begun the process of converting people from the internal system to the new commercial one. As well as interviewing a variety of developers, project leads, architects and senior managers, I was able to attend a class where developers received their initial exposure to Tool Corp.'s product.

This study consisted of a series of semi-structured interviews with managers, configuration managers, and developers. All the developers interviewed had used the new tool, and many of them had used the old tool prior to that, or some other CM system. The semi-structured interviews ensured some overlap with the dependency management problems discovered at Tool

Corp., and provided opportunities to revise and extend understandings of how dependencies affect software development. The interviews themselves lasted from 30 minutes to an hour. Over the course of two days I conducted a total of 13 interviews, and attended a class designed to introduce developers to the new system. I also gathered public documents about the company.

5.2 Individual Level Dependencies

Parallel Development Dependencies

The developers at Computer Corp. like those at Tool Corp. engage in parallel development. Also, I found the same concerns about the practice as I had at Tool Corp. While the developers at Computer Corp. might have been experiencing adoption problems with this particular merge tool, those at Tool Corp. were familiar with the technology. However, because the developers at Computer Corp. were new to the merge facility they revealed important aspects of parallel development dependencies.

As the following quotes demonstrate the developers at Computer Corp. had very similar concerns about merging and parallel development as their counterparts at Tool Corp. The relationships between the parallel pieces of code often required deep levels of understanding about the behavior, purpose and implementation of the modules:

> But the problem is if you don't really understand the code very well if you don't understand that module very well you don't know if you're having conflicts that have to do with design unless you really understand the module well so that's why it's bad (1: 328-331)

The scale of operations at Computer Corp. means the developers often find themselves in parallel development situations either with several other developers all working on the same file or even working on multiple sets of changes on the same file themselves. As one developer explained,

> Oh yeah, um, it has to do if there are more than two versions out there that are in parallel we have to make sure that what you're actually merging is right one. And if you have a three-way merge then you have to merge the two and then the result of that with the other one. So it's a three way merge. It some way it will if it's tagged it will be easy or sometimes you make comparison between the two, you do compare, diff between the two files. And then if you know that the line of code very well you would say, oh these are the changes that appeared, especially if it is um you know correctly labeled I shouldn't say that um commented inside the source then yes. It's all the process is still a big pain because see putting that comment inside the source is still a process you have to make sure you say these are the changes I did for this functionality and then I will say this is the change that I did for functionality number two so that you can actually merge the two together. (6: 1534-1543)

Tagging the software so that he can merge it together in the right order turns out to be a complicated part of merging. Even when he made both sets of changes himself the context involved in understanding both modifications often got complex enough to be confusing and disorienting during the merge.

One of the developers also described the ramifications of getting the merge wrong on the entire life cycle of the product. As she described it,

> It's too easy to make mistakes, it's easy to introduce errors and that's so late in the process you don't really don't want to introduce errors that late. Cause its' you know it's right before you're going to code freeze or start testing or something like that and well and one of the problems with merge is you have to really understand both parts of what it is that you're doing so you have to understand all the stuff that you're merging and naturally if its a merge its one person or another person wrote it, so either you both have to do the merge together which happens sometimes or um or one person has to merge but they may not actually understand all the code changes that happened before them to understand the ramifications of what they're doing so that's my feeling on it. That's why I'd rather avoid them. (1: 318-326)

Code freeze refers to a point in the life cycle when no more changes can be made to the software. This precedes a period of intense testing, where bugs and errors surface. The code may then be altered, and re-frozen, and re-tested. Often this cycle occurs a number of times in the hope that the product will emerge with fewer bugs and be stable enough to release to the public for sale.

When any developers reach code freeze and testing, they become much more concerned about the changes that they make to software. I witnessed this at both Tool Corp. and Computer Corp. The test and fix cycle creates a great deal of pressure on the developers, usually they have a major product deadline coming up, like product release, and the first cycle often reveals many bugs. They work long hard hours trying to integrate and re-test the code. At this point, parallel development creates even more stress, because it requires that developers realign their efforts with each other, and cuts into very tight schedules.

The developers at Computer Corp. have a number of strategies for coping with their parallel development situations. The developers at Tool Corp. often discussed the poor division of labor that they believed created the parallel development problems. At Computer Corp. the development teams felt that they and their managers tried hard to divide the work up so that the product had conceptually distinct areas:

> I guess there are at various times 2 to 4 people working on the model at the same time and we attempt to separate it so that we don't step on each other's toes too much I mean that's makes it easiest if we just parcel up the work in such a way that we aren't using the same bits of code (1: 272-274)

Unlike the developers Tool Corp. the developers at Computer Corp. realized the limits of a good architecture. Software that gets divided into conceptually distinct sub-systems may still have a sub-system that ends up being more central than the rest of parts. This piece of the system, the kernel, often ends up having many dependencies between it and the other sub-systems, which means that people working on kernel software must maintain more relationships with other groups than any of the other groups. As one senior manager explained, this creates an added complexity for kernel software development:

> We'll there's the [kernel] people which we which don't use right now they have as many as 30 people in one file at one time. They'll have might have their whole project team in one file. Well, it's historical there files are fairly large and because its a, because kernel has to manage so many resources ... no matter if you think even if conceptually the problem looks separate its it all has fingers back in this one central kernel process its and so many times you have multiple people working on one file and that's an extreme case and definitely out there to some of problem but it is a problem we have encountered. (5: 1236-1242)

A good software system may have a central kernel that orchestrates the functioning of the entire product. Dividing the labor into the different sub-systems works well for the people who work on peripheral sub-systems, that may be less connected to other sub-systems or only to the central kernel. It does, however, mean that developers working on the kernel need to manage the possibilities that many other people have the very same file checked out for different changes at the same time.

Developers use code reviews to find out whether the work of others may impact their own:

> Oh yeah, that's always hard um the way we do it here is we share technology by doing mostly for example you actually join the design process from there you know what should be incorporated in the source and um other one is also to um to code review when we code review that's when we actually know the insides the internals of what a person has done and from there we actually manage to learn more yes that helps. (6: 1546-1551)

Team meetings of any sort, for example: to discuss design issues or analyze fragments of code have critical social implications for the management of dependencies. While developers recognized this for parallel development — they saw these as opportunities to find out what modifications other people were making to various modules — these meetings help to manage other dependencies.

The developers at Computer Corp. use a variety of tools to help them in their configuration management functions. The corporate policy sets out a trajectory to move all the developers to Tool Corp.'s product, but during my visit the developers also used Revision Control System (RCS) a UNIX version control product, and a more complex home grown tool that I call "Alpha." I will discuss the conversion trajectory later on in this section, and where necessary I will make a distinction between the different tools used.

The developers used these tools to support them in their parallel development work. Those who used RCS had to perform manual merges using editors like vi and emacs, and the UNIX "diff" facility as RCS does not provide a merging facility. However, like the developers at Tool Corp., and perhaps heightened when using RCS doing manual merges, the developers viewed merging as a difficult and time consuming situation to be avoided. One developer described his need to rush to finish his work:

> Because I think even in there's no easy way to merge if somebody changes some code will detect it but you still have to go look at how your code is going to effect the changes that were there and how it integrates in there. So really everybody wants to be the first one to have the code and then everyone else can merge laugh I means that's the but I don't see any way around that. (6: 1958-1961)

However, for some developers this rushing to finish first and avoid the merge presented a new dilemma to them regarding the quality of their work:

> We're at odds we're in one of these catch 22's again. It slows down our basic premise is check-in fast so you don't have to merge cause this is not what the model is trying to encourage us to do? That's why I don't like it (because it encourages that) I don't like this encouragement to check in first I mean granted OK it will only complicate and OK hey I race through the code get your changes done but then you go to the other model that says you need to take your time and do a downtown job we need to take these steps to ensure that your code is maintainable, la la la, code reviews all these source inspection things, so I'm not sure that I like a computer system that rewards short cuts to the standard models of software engineering practices. (7: 2223-2230)

In this particular interview the developer had hoped that the new tool would reaffirm his own priorities of how good development takes place, by code reviews, adequate testing and planning for maintainability. However Tool Corp.'s product does not give any priority to merging and in the absence of that, typically the merge gets left to the last developer to finish to do.

Another problem that the developers had with the new tool was that it did not appear to accommodate their current working practices. The fact that new tools require adaptation, and that they involve changing work practices, is not a new discovery. During a my visit to Computer Corp. I attended a class for developers new to the tool. The intent of the class was to explain how Tool Corp.'s product worked and how they could use it in their own software development work.

When the developers discovered that the tool did not let them save a semi-merged module they got quite concerned. Alpha, the in-house product that they had been using, offered a different paradigm for merging. Typically with Alpha the developers checked out all the code, the entire sub-system, and then merged the entire sub-system back into the main line of development. Rather than merging specific code modules, the developers merged large sections of the product together. Under this paradigm of thinking about merging, a semi-merged state seems obviously necessary. They applied the Alpha merge model to the new tool seamlessly and it was up to the

class teacher to explain how Tool Corp.'s product required a different merging strategy based on a lower level of granularity. The teacher's paradigm was "merge little and merge often."

At Computer Corp. parallel development, and merging, happen in different ways according to the tools used. These tools influence developers thinking about how merging should be done, how often, at what level of granularity (a module or an entire sub-system) and consequently influences their strategies for coping with the merge itself. When the tool changes the dependencies that created the merging situation in the first place need to be dealt with in a different way. Developers can not use the same strategies for merging an entire sub-system as they would for merging a module, and so the change to a new tool means revisions in the ways that they cope with parallel development dependencies.

Change Dependencies

The developers at Computer Corp. must also manage change dependencies. Despite the differences between the methods of assigning changes to developers — Computer Corp. did not use the same problem reporting facility as Tool Corp., and appeared to have a traditional manual procedure in place — the developers still end up depending on getting all the parts of a logical change together.

The size of the product being built at Computer Corp. implies that technical change dependencies may diffuse across the product, and as a consequence across the organization. As one project manager put it,

> Um, well in particular since much of the kind of software I work on um and the people are associated with me work on, is um, infrastructure software and has lots of dependencies on lots of other bits and pieces of infrastructure um and bits and pieces of operating system um because of that there are lots of issues on trying to keep data structures in synch trying to keep changes in synchronization um as well as the general issue of supporting multiple threads you know because we um because of the customer driven requirements of being able to support multiple releases we wind up with multiple threads going on in parallel and you want to make sure you get all of your fixes um coordinated across all of those threads. (9: 2705-2712)

At Tool Corp. the developers usually talked about these changes in both technical and social ways, describing changes in other sections of the code and who was working on them. At Computer Corp. the developers relied on the other sections of the code being accurate, and the function of ensuring that all the changes worked together was handled by the build manager typically. Change dependencies thus were managed like integration dependencies at Tool Corp. at the point when the product was assembled.

The build managers that I spoke to recognized the importance of managing change dependencies. One build manager described her role as build manager as:

> It's my role to make sure that everything gets into that's supposed to and nothing gets in that not supposed to do. Notifying everybody when I update it and that sort of thing (1: 472-474)

Another person responsible for the build management function in his group described the inability of the tool he was using RCS to track these changes. To get round the short comings of the tool he maintained a manual system:

> and unless you have um a system for doing that which people have done in a manual way like writing down on bits of paper talking to 18 different developers ... (9: 2720-2723)

The developers at Computer Corp. face additional problems to trying to make changes to the code due to the size of the development effort,

> I suppose our most recent one last summer um I probably spent two weeks um just trying to reproduce an environment um and make a one line change. That's correct. And that's, and that is a perfect example and it's something that I see time and time again happen. (9: 2697-2699)

Even though the developer wanted to make a one-line change he needed to ensure that it worked with all the code that interacted with the revised module. At Tool Corp. the tool provided the developers with a stable base from which they could all begin working on the related changes. Developers at Computer Corp. did not have this luxury, and so they reveal another aspect of change dependencies, that before any changes begin the developer or developers have to find a working version of the product, a baseline from which to start making their changes. As well as being dependent on other developers working on other parts of the logical change and those who assigned the change, they also depend on those responsible for all the parts of the code necessary to form a stable baseline from which the changes can be made.

As well as finding a stable baseline from which to start making changes, developers must synchronize their changes:

> then there's areas where the developer doesn't really know who the other developers are very widely separated by organization and yet they work on co-dependent code and so that communication happens through um what they call [steering committees]... and there's two or three different kinds and there organized around a release ... so if a change gets made in a lower level routine it could impact a lot of people and so there's formal notification techniques ... we use e-mail and some more informal things where you just know because you've been around a long time. You tell everyone that you know who would be impacted that there getting impacted. (5: 1263-1276)

The developers at Tool Corp. relied heavily on the tool to help them synchronize their changes. As a small group using the tool they were easily able to work together, and even find the other developers (located in the same building often a few feet away from each other) working on

those changes if necessary. As the developer quoted above describes, the situation is vastly different at Computer Corp. When the developers work on changes that span different sub-systems, they find themselves working on modifications that cross different organizational boundaries. This is reflected in the need to create committees to guide the changes, and communicate the dependencies between the groups. However, at the same time, the developers use electronic mail, and other more informal communications, such as finding the old-timers, to help them negotiate and coordinate the changes they make with other distant developers.

Change dependencies, like parallel development dependencies, were also in the midst of transforming with the adoption of Tool Corp.'s product. An Alpha user described his concerns about how change dependencies would be managed by the new tool:

> Um.. if its um you have a pool of versions to choose from and you have one user down here and another user over here and they're in the same project, the same set of things, um, lets say this guy over here makes a change to a change C1 and he has a new module which means he had to change the makefile... and over here this guy is just about done with all the stuff most everything um its almost five o'clock and this guy is just about done and he says OK I [check this in] and make it available the makefile this I think is OK. Um meantime [the first guy says] OK I'm ready I want to get the latest and greatest of everything OK um again because we're in a pool of available versions here and it's not project oriented he gets this and his build that night fails. Cause he doesn't have the complete set, he doesn't have a valid configuration. (4: 941-952)

Alpha used a project — a collection of modules — as its unit of operation. Tool Corp.'s tool, like most modern commercial configuration management systems, emphasizes the module as a unit of operation. At the project level Alpha users thought in terms of receiving sets of related software all at once, and the new paradigm, required selecting these sets module by module. As the developers shift from Alpha to the new tool they will have to rethink some of their strategies for managing change dependencies; in this case, they may have to be more alert to the possibilities of related changes instead of depending on Alpha or the build manager to take care of that.

Expertise Dependencies

The developers at Computer Corp. relied on the expertise of their colleagues to assign work and to help with their software development efforts. When developers work on sections of the system that they do not understand they often rely on their colleagues' knowledge of that code, as this developer told me:

> we pretty much design not on our own and with bug problems if it involves more than one sometimes bugs right now since we're doing a lot of bug fixing um a lot of them you know immediately or you can fix them immediately but then the ones that involve multiple parts of the system then that takes two or three people to figure it out sometimes. (1: 537-540)

Expertise itself often ends up being the way to determine who does what work. Instead of the relationship between different sections of the system defining the social relationships based on expertise, developers' knowledge means that they end up working together:

> Any time we do something that's design sometimes it's all of us and sometimes it's just two of us but it's almost always not just one person and then we come up with a list of tasks and then depending on what our expertise is each of us gets assigned different tasks some of us are experts in similar areas so one or two people could be assigned depending on what our work load is (1: 287-291)

In this group two developers may get assigned to the same task because they both have the required expertise. This takes them into a situation of having to work together very closely as the work of one will certainly impact the work of the other.

Integration Dependencies

At Computer Corp. each team builds their own sub-system routinely. One developer usually assumes the role of the build manager, as I have already discussed, and takes responsibility for doing the build. These developers spend some considerable time and effort coping with integration dependencies, in the absence of having automated mechanisms of pulling together the latest changes from all the developers. Unlike the build manager at Tool Corp. the build managers at Computer Corp. used manual strategies for gathering latest changes:

> So it's not everyday it's on an as needed basis and I make them tell me stuff like what's object in my sources to grab and what's the version number of them what bug are you fixing and how am I supposed to know once I install it if that bug got fixed or not. What's the behavior I'm supposed to see and then when it's necessary I do the reconfigure and install and send out mail. I get developers to tell me that my mail or in person it depends on whether it's a big one or little one. Usually if it's more than just one or two minor things they send me e-mail but if it's one or two minor things then they just tell me and I write it down. I keep a track of it. And how I track is that every time I do an install I send everybody e-mail saying this is what I installed this what is was supposed to fix and it's ready for you to use now or whatever and so that part of it I have to keep track of. (1: 486-495)

However, as the build manager went on to explain, these manual strategies have weaknesses:

> The dependencies are basically held in the knowledge of those who are developing the software... really there's really no good way to record it anywhere ... the only place where you can really find it is in the build instructions of the makefiles you can see some of it but it's not clear there what all the dependencies are ... a lot of it is manually specified so if they're wrong they're wrong. (2: 600-604)

Although she can track some of the technical aspects of integration dependencies, using build instructions and makefiles, she still relies on the manual specification of these relationships. When developers get that wrong, then she has to take the time to find out where the problem is and fix it. This involves communicating with the developers and finding out what they know, and then building the whole from the parts that each developer understands. As one developer put it,

> Um actually a build manager is a big role, it's one that knows mostly the structure of the system especially with I would say because um you really have to know like in our practice here once I have checked in ... a version ... to a higher level of visibility something that the build manager can see ... she still has to ask me what those versions I have checked-in so that she's sure that it is really picked in the configuration that she has so that's the kind of role our build manager plays right now. The build manager really has to know the structure of the system and what changed what needs to be there what is really missing if in case there are any. (6: 1631-1638)

At Computer Corp. integration dependencies take place at different levels of operation. Developers assuming the build manager role typically assume responsibility for gathering components for a sub-system of the over all product. They are part of a small team of developers working on a similar section of the product.[22]

However, some of the groups have complex integration dependencies. In these larger groups no developer can both develop software and assume the build manager role, and Computer Corp. provides an organizational function instead:

> what happened was for hard configuration problems specialists were identified, so we had these in certain areas people who job title was [build manager] and that's what they do they manage configurations mainly because they're so important that they knew that they had somebody to do it but no developer had the time to do it. (5: 1183-1186)

One of the reasons why integration dependencies get difficult is because the size of the group increases. Some sections of the product, often the central sections like the engine and the kernel, have over thirty developers. The developers find it impossible to integrate all the changes and continue their own work, if the team is large:

> I'm not a big supporter of, I don't think there are that many places at [Computer Corp.] anyway where there is a build manager. There are some. There are some large groups for large projects that have build managers, [builders] are responsible for building the release kind of thing, but I believe there are more groups that don't have that. (4: 1051-1054)

[22] Small teams have no more than 20 developers in them.

I call the people whose job was build management "builders." Computer Corp. had around 50 people, in comparison with the 700 developers, whose job function was:

> Our [builders] currently do a couple of things that are important one is that they actually do the final build and release if you have a [builder], now in our group we're so small that we provide our own [builder] function but when I worked in different groups that were larger we had a [builder] who did that for us and that was ideal in terms of when we said we were done and we released our code and they captured it they put it some place and then secured it so we couldn't get at it so there was no more last minute sliding in, when they said code freeze it was code freeze laugh yeah code freeze or die so I do like that I like the I mean I again its sort of like you get used to the concept of code freeze date meaning something and you are tested you're ready to go you know go ahead take it. (7: 2258-2265)

As well as managing the integration dependencies of teams, builders also provided other critical functions as a result of their role. Their ability to enforce code freezes, to prevent developers from modifying the software, and other policies, meant that the developers could rely on the system in ways that groups without builders couldn't. This is similar to the ways that the developers at Tool Corp. rely on the tool.

The builders at Computer Corp. helped individual groups to integrate their sub-systems. At the same time, along with the release area group the builders provide a broader function, as senior manager explained to me:

> Literally it takes, there's if you look at the people involved it takes a lot of people just focused on their work problem. No one person can actually release anything it takes 30 people in this group to the release area and 20 people and 10 people in building and 10 people in that people and what is their job? What do they do? They just going around trying to figure out what the configuration is, and how to build it, and why this build failed, that's all they do (5: 1377-1382)

As well as having large teams of people, the organization needs to routinely integrate the entire product together. Builders begin this process by gathering individual sub-systems together and trying to figure out what other parts of the product their sub-system relates too. Especially as the organization moves close to the release date they need to be able to construct the entire product and test it out together. As one manager told me,

> It's a big problem now to, the big problem is when things build correctly there's no errors and yet they hit the field and there's incompatibilities because not everything was really integrated at the same time and that's the key gotcha in all of our process. (5: 1288-1290)

It was this problem, building the sub-systems, that lead me to call these dependencies integration rather than build dependencies. Software engineers are familiar with build dependencies in a technical sense, but build dependencies typically refer to the linking together of components in a sub-system, something that a makefile resolves. I wanted to capture something more than that,

not only the social aspects of the low-level build dependencies, but also these higher-level integration issues.

Because sub-systems must come together to create the working product, Computer Corp. must find ways of managing the complex relations behind the code itself. Currently Computer Corp. does this by having an organizational unit, the release area, managing all the relationships between developers solely through the code:

> because right now development does all of its things and then it throws over a very high fence to release and then release has a whole other different process to managing these configurations, and literally they have only one version at any given point in time and then the next release window happens they get an entirely different version and if the first one is completely overlaid. So development can never go to the release area and say give me something from last week because it's not there in release. It's not there. So if we could get a version scheme into release then we could get the release to be more like flexible in how they deal with releases and hopefully speed the throughput because right now there's release windows and you can only release during those times cause that's how they manage their versions. (5: 1362-1371)

Computer Corp. relies on the release area people to construct the product from the code they have at hand. If the release area people have problems making the product fit together then they notify the groups whose code does not work. This comes with a price, as the quote reveals, the release area people do not keep older versions of the entire product, so developers can never go to that group and get those older versions back. They have gotten re-written by the release group, and are very difficult for any development group to construct without help.

Integration dependencies reveal an important difference between Computer Corp. and Tool Corp. To resolve integration dependencies between developers Computer Corp. provides special people within the organization, builders. Computer Corp. also has an organizational division, the release area group, to work on a level of integration that Tool Corp. does not deal with at all. What one developer can do in their spare time at Tool Corp. requires an entire department at Computer Corp. As the scale of development increases, the complexity of dependencies rises. I will return to this in more depth later.

Historical Dependencies

Like their counterparts at Tool Corp. the developers at Computer Corp. also relied on decisions and changes made in the past; however, unlike Tool Corp. Computer Corp. did not have such a comprehensive organizational memory. Most configuration management tools provide some way of viewing the evolution of software over time. Even RCS, one of the early versioning tools, provides developers with past versions of code and a chance to enter some comments about actions taken on a particular version of they want to.

One developer described a particular instance of going back to examine code in the past:

> Well um the particular example I'm thinking of that helped avoid a mistake was um where I was able to look at the [evolution] of um a set of changes to a module and um graphically compare them and that was be able to do this in an almost instantaneous way um is very helpful cause if the barrier to um like a cognitive barrier if you have to go through a lot of steps to figure out the history and do comparisons and things like that you tend not do that. You know cause I don't have time to do this, I'll just kind of blast ahead and cross my fingers and hope that I haven't screwed up. That's the result, OK, if you can easily by clicking a few buttons, um get the comparison, see the history, and instantly see why this particular change was made um how it does or doesn't affect this next set of changes or you know understand why um this person made this change and er hadn't considered this other problem and that sort of thing. (9: 2733-2742)

The developer used a combination of tools to help him see the differences between the two versions of code that he was interested in because he did not have access to a merge facility. He then had to work out what those differences in the code implied, what was the change that caused those particular differences? Sometimes however, this proved too time consuming and the preferred mode of working was to hope that his general knowledge of the software would ensure that he didn't break anything while making his changes.

The developers who had switched over to Tool Corp.'s product found some difficulties with the way that the organizational memory worked for them. Even though it may have reduced the time spent figuring out why two versions differ, it presented new challenges. As one developer explained to me, using a Presidential metaphor,

> Yes, I do I mean I've used the [evolution] command a lot if anything that's kind of valuable I wish by default it would go down to the last one though versus the beginning one. Because I don't really care about history it's like it's like saying who's president now, George Washington or Clinton, I don't care about Washington because he's way back there right now and I need to see where Clinton's going right now because that's affecting me right now today whereas the source control I need to see the one that's latest in there, I don't care about 1.1 where not using 1.1 we're on 1.50 or something depending on what file so 1.1 means nothing to us. Oh sure I'd go back to Bush Reagan or Carter. (6: 2014-2021)

It was only a simple request, that the tool display the nearest ancestor rather than the furthest ancestor when invoked, but when the software evolves through several hundred versions, still showing this distant relative will be very frustrating. It was also symptomatic of a bigger problem that he described:

> Finding out the information like um what functionality I want to add to my next release is in that [evolution] view and sometimes looking at versions looking at what people are doing with you. Yeah, it's powerful in that way um you can also sometimes if you want to rearrange although it is created in a timewise manner sometimes you still want to rearrange laughs history and I found myself doing

> that because one object needs to be in front of the, it's more the functionality within it needs to go after one object that have created from before, they're not really even though they're serial or and yet they're not really serial, anyway, yeah I rearrange things at times. (6: 1571-1578)

Software visualization is an important research topic today. As this developer explains, sometimes even though history is the appropriate context, the serial view of the evolution is not the most appropriate. The developer wants another way of visualizing the evolution of the module, perhaps following various changes as they evolve. The developers at Tool Corp. never mentioned reorganizing history, and I speculate that they have adapted to thinking about software evolution serially. However, other factors such as the size of the development effort may also impact the strategies that the developers at Computer Corp. use to manage the historical dependencies they have with others.

Configuration Management Tool and Practice Dependencies

The developers at Computer Corp. use a variety of systems to support their configuration management activities. I have already discussed how they come to rely on those tools. When they are changed the developers must find out what the new tools support. The developers then have the choice of revising their configuration management practices to fit the tool or ignore the new system. However, the developers that I spoke with at Computer Corp. generally valued any support that they could get for configuration management, and along with their interests in technology as software engineers, they usually made some effort to try to accommodate the new tool. In this section I review some of their configuration management dependencies not discussed earlier.

All configuration management systems, from the earliest versioning tools, support some mechanisms for backtracking. Simply by storing versions of software the tools allow developers to go back to previous versions when they make errors:

> Being able to go back in time is nice because providing I know exactly what I can go back to, I can say add a path as existed at this point in time, so I have all old modules even leading in the past, the future from this point, even in the [operating system] they're there so I don't have to, I never have to have any of these worried about. If I really screw this module up I delete it, but I can delete it, I can do anything I want to, because it's still there. (4: 1110-1115)

Developers like this feature simply because it provides them with security, a knowledge that whatever they do they will not lose something that works. It encourages them to use configuration management tools in their work.

During my visit many of the developers that I spoke to were changing tools. As I have already mentioned, the tool change was forcing developers to adapt their strategies for managing dependencies. At the same time it was forcing them to revise their perceptions and understandings of configuration management and as a consequence of development itself. As

one developer told me after I had asked him questions about his configuration management strategies,

> I guess maybe I'm just real fond of RCS laughs I mean its one of those if you've use it for a lot years its like the devil you know so I'm trying to like get out of that mode of thinking and into this new mode well no I mean like you're bringing up all my fears and worries and concerns and all the things that I'm still struggling to try to get into this new paradigm and this new mode of operation and I'm having and I'm going through the learning curving, I'm going through the learning curve of a completely different method of operation and I'm not happy about it. (7: 2179-2185)

As well as developing their own individual strategies to cope with the changes, the introduction of the tool creates situations where developers need to devise new group practices. The same developer also described how he and his colleague, who were working together on a small section of the product were trying to devise policies for using the tool together:

> [Developer A] and I are trying to come up with our own set of operational procedures and I mean its sort of working but its not a corporate wide policy we asked [Manager] at roll out time what I thought was a real elementary question, how should we name our projects, and we don't have a standard for that, our projects, like we have um product numbers so all of our products have [numbers] and it sounded to me like maybe all of our projects should be named based on the product [number], release well if you go in and look at our database laughs yeah you would see that it is just hodge podge of we have no uniform rules we're still struggling with this, we're taking big steps. ... I think that my opinion of a roll out of this should have been that there should already been a set of rules standards that our company someone said here's how we want you to operate instead we're like learning on the fly um like everything else we're doing it ad hoc. (7: 2237-2247)

At Computer Corp. those individuals responsible for helping the company to move to the new product were very busy. Often they did not have time to spend helping individual developers, or groups, work out policies for tool usage. This left the developers wondering how to incorporate the tool into their work routines.

Platform Dependencies

One dependency that the developers at Computer Corp. encountered that did not affect people at Tool Corp. involves different hardware platforms. At Computer Corp. they develop their product for one particular hardware platform that Tool Corp.'s tool does not run on. That means that the developers working on the unsupported platform must develop their software on another platform where the tool runs, and then cross compile their code and port the compiled versions onto the unsupported platform and then run it.

This creates additional work, as well as additional complications for using the tool itself. Because multiple developers need to port code they have had to work around some of the features of the tool itself. As one developer explained,

> we ultimately have to move our source code up to [unsupported platform] and ultimately do our testing and building for our release product up there but to shorten the cycle we use cross-compilers on the Sun. [Unsupported platform] does not allow [Developer B] and I to share a directory unless we share the UNIX directory [and] Tool Corp. tool does not allow us to have two different projects that both have the same [directory] ... seeing as we both work in the same office we said lets share the Tool Corp. [view] for right now and we'll figure out later when they integrate fully with how this cause its not clear right how we're going to do this. ... We had to change all of the permissions on Tool Corp. tool ... basically anybody can do anything right now to our project we're running in a very um unprotected mode of operation. ... So we're running in a strange mode where this is not ideal but until we understand the integration that we need to go through with um which they don't have yet and it's like it's always coming its coming any day now, it's coming any day now, er until we get that straightened out. (7: 2110-2127)

Every hardware platform has certain unique features. These features both liberate and constrain developers in their work. They may provide developers with opportunities to improve the software, for example make it run faster on that platform, if they exploit the features of the hardware. At the same time they constrain developers, because sometimes these features dictate how certain software must be written.

The developers at Computer Corp. build software to run on some platforms that Tool Corp.'s tool does not run on, because they offer certain features that are important in Computer Corp.'s market. At this point the developers invoke technical relationships, between the platform for development, and the target unsupported development environment, where the code must run. This changes not only their way of working individually on the system, but also ways that they work with each other.

5.3 Scaling-Up, "Group-Level" Dependencies

Life-Cycle Dependencies

I have already described how at Tool Corp. different platforms have different release cycles. At Computer Corp. the developers must also manage those kinds of differences. As an organization however, they have multiple life cycles going at different levels as well. In individual groups the developers work on a life cycle to build a specific sub-system. At the same time the entire product has a larger life cycle, the time to release cycle.

> Then there's a bigger life cycle because within that product now they have to come together and within that coming together all these products then they'll got through and that coming together all the products are actually more in the QA life cause that's when you integrate everything and you test the integration of all these things. It's big it takes um right now they they're as far as productivity's concerned they have been releasing things within like 15-18 months which is really short laugh but before it took two years. (6: 1622-1627)

Simultaneously the developers must work towards completing their product as well as the entire product. Sometimes groups do not contribute to a specific release as their section of the product stays the same between two releases. So at any time, Computer Corp. has a number of life cycles going on, with different groups always involved in their own small sub-system life cycle as well the product life cycle, as well the different product release life cycles.

"Big Picture" Dependencies

Trying to understand the "big picture" is virtually impossible at Computer Corp. The product as a whole contains millions of lines of code, that no single person could possibly remember and understand the relationships between them. It would also be impossible to try to imagine all the states that the software may get into during operation for a system the size of the one that Computer Corp. builds.

However, the developers at Computer Corp. still need to understand the "big picture" of their own sub-system. As one developer described the problem,

> Because, OK, I mean it's easy if you see these couple lines for instance if they just incrementing a pointer or something like that. But if those couple lines for instance change the state in a sense and that state may reflect may affect you and what you were doing because now you changed the state a different way or you added another state so if you had the state diagram before in the previous thing where you had four states but now after you merged it you see that oh god this thing has ten states cause they added four more to them it's like well can I actually do the changes that did in this state? Right, so that means you don't understand exactly what happened yet until you examine the whole code you may actually just see oh well wow there's four different states now added to here but that means nothing to you have to look in the context of the big picture. (6: 1964-1972)

They need this kind of information for a number of reasons, to cope with parallel development as well as to make modifications and understand how those changes will impact the work of their colleagues. As a team of developers working on one sub-system they struggled to maintain a vision of what their part of the product did.

The scale of their sub-systems easily matched the size of the entire product at Tool Corp., and as such these problems were comparable. However, I noticed a marked difference between the expectations of developers at Computer Corp. and those at Tool Corp. Developers at Tool Corp. felt worried about the fact that they had lost the ability to see the product, and behind closed

doors they blamed bad management practices for this. While individual managers may have played a role, it is more likely that the loss of the "big picture" accompanied the expansion of the group in size. As the group grew bigger more people worked on different parts of the software and the developers slowly lost track of what everyone else was doing.

At Computer Corp. while developers wanted to understand the big picture, they did not view the fact that they couldn't as a managerial problem. As this developer put it,

> I think in general in many ways the kind of work that I'll call trailblazing or frontier mentality um the time for that has gone. Most of the kind of work that can be done by an individual now here's this well-defined thing and you take this you know asocial or anti-social individual and put him in a room you know lock him up throw raw meat under the door doesn't work anymore because most of the work that at least for [Computer Corp.] and I suspect that's true in just about the whole industry most of the work that could be done under that paradigm has been done and now all of the new things have lots of dependencies, lots of variations, systems upon systems and you can't make things like that using one person cause one person can't do all the work can't see all the ramifications can't understand the whole system, its just too big for one person to mentally grasp and so you have to do things with teams and so it says the kind of person that didn't like to work in groups is a dinosaur. (9: 2769-2779)

For the developers at Computer Corp. the lack of understanding the big picture was simply a consequence of working in large teams rather than as individuals. It was also a consequence of working on a large product rather than a small one. The developers did depend on their understandings of the "big picture" to make decisions about what course of action to take. However, unlike the developers at Tool Corp. they were resigned to the fact that their own interpretations of that big picture were incomplete and possibly mistaken. Perhaps because the release group or the builders took care of the final integrations of the product the developers at Computer Corp. didn't worry too much if they made mistakes based on incomplete information.

I have already discussed two organizational functions that Computer Corp. uses to cope with dependencies, builders and the release area group. Computer Corp. also has another class of employee who manage "big picture" dependencies that span the different development groups, architects.

At Computer Corp. architects map out the higher levels of the product. They work hard to define the overall product structure, and then get involved in breaking down the sections into sub-systems that different groups can work on. Because they usually start from existing code, the architects often find themselves in the position of mapping out the new extensions and directions for the products, in technical terms, and then assigning the work to the appropriate groups. As this architect put it,

> as a system architect there are not a lot of architecture paper generated, there are some I think, in fact there's a discussion going through most of the job isn't so much er thinking up new architectures but getting them accomplished. ... that's

> the larger part of the effort ... in many ways the act of coming up with a specific architecture is you know just draw some lines and boxes and arrows and it's almost a dime a dozen. Lots of people have um intellectual architectures and not too many people can translate those into actions and um agreement and creation, that's really where the rubber meets the road. (9: 2816-2824)

An architects work consists of the translation from the direction for the product, into logical changes required, into technical changes that need to be made. At the same time, the architects I spoke with recognize that they also manage these dependencies that they define and create for the groups. They become boundary spanners, helping groups communicate with each other. As the architect explains,

> Oh, well um, to give some examples um without naming names there have been cases where say one group has been at odds um they have and largely driven by different role leaders, each one kind of thinks its the center of universe and um they have totally different management chains um but they have a dependency relationship, right (the code has the relationship but the management chain doesn't) and over the years there have been you know repeated friction's and um you know at times they as a result of them um mail wars diatribes um people yelling that kind of thing and I think that's behind us but that kind of thing can happen and um when you do have a pair of groups then who have this dependency or interdependency it becomes difficult to accomplish change and when change is really needed in order to deal with um technology advances with competitive requirements things like that um we need to kind of revisit the interfaces between the software components that those two groups produce and as a result we need to revisit the interfaces on a personal level. (9: 2791-2802)

At the same time that architects define, create, and sustain technical dependencies between the different groups involved in building the product at Computer Corp. they also engage in social dependency management. They move between groups presenting each with the bigger picture, an understanding of what the other group does, and why that's important. Garlan and Perry describe these social properties in dry terms, but for practicing architects, conveying these meanings, building these shared understandings, does not happen as a result of drawing diagrams, instead it involves spanning different groups and artfully persuading them to fall in line with the main development efforts.

Shared-Code Dependencies

I have already discussed the problems of crossing groups, in the discussions of specific cases of dependency management. Sometimes individual developers find themselves working with other developers remote to them, managing changes. Other times individuals either in the development group, or specially assigned individuals must integrate sections of the product together. A broader, group-level problem, concerns shared-code dependencies.

Modular decomposition begins with an idea for a system. The idea gets broken down into small conceptual units, and through principles of modular design, becomes sub-systems ready to for developers to implement. During the process of modularity the emphasis focuses on "separation of concerns" dividing up conceptually distinct parts of the system. For example, it would be bad practice to confuse kernel functions with the user interface. Even though good software engineering practices focus on the distinct modules and sections of code, the links between them remain.

The links between different sub-systems cause developers to need to access the code written by other groups. Shared-code dependencies arise when one sub-system relies on another to function. When shared-code dependencies occur then the groups involved must work together enough so that they get the software that they need to complete their own work.

One developer at Computer Corp. felt that the problem was particularly bad within this particular product because it was designed to have many shared-code dependencies. He said:

> Now partially you know the way other companies deal with this is they just they structure themselves so they don't have to ... um if you look at [large computer company] for example ... the way they tend to do things is much more layered. ... the reason our computers are so much faster in you know these in our targeted applications is because of that kind of integration for example [database company] can't hope to compete against our SQL product because our SQL product is tightly coupled to the operating system. [Computer Corp.] avoids that by you know our SQL people will talk right to their counterparts writing disk drivers and operating system message passing services and all that. So that's well and good for performance but then we get all of this big ball of yarn tons of co-dependencies instead of nice clean message interfaces right. We have different er parts of the software that have to intimately be in bed with other parts so it's a big mess (8: 2384-2403)

The technical trade-off between performance and layered code in his opinion had created more shared-code dependencies, occasions where during operation the program jumped between sub-systems to increase performance.

Shared-code dependencies also have a social component. To ensure that the sub-systems operate together developers from different groups need to have access to latest working versions of other parts of the product, so that they can align their development and test their own code. They rely on the other groups to put their code somewhere where they can find it. This turns out to be more problematic than it sounds:

> The issue that [Computer Corp.] developers face today is they'll put a location in their build file to point at a piece of shared code and it might change locations it'll move and reason it moved is because a new version came out so to ensure that no-one uses version they'll delete the old one and move it to a new place but this poor guy doesn't know where the new one is so then its research and rehash and time delays cycle loss... (5: 1345-1349)

Developers have a number of ways of trying to find and maintain their knowledge of where that code resides. One developer explained how he managed his shared-code dependencies:

> it's all word of mouth, you know have to just go to this person who's been with this group a long time and has a great memory and say who uses it, where does this thing come from you know, so it's pretty crude (8: 2497-2499)

However, this method of tracking shared-code has a number of problems. If the person who's been with the group for a long time leaves then the developers need to find some other person who has a similar network of contacts across the organization and the institutional memory of what ended up where.

The other side of the shared-code dependency, having another group relying on your code, also requires management. If development groups leave their code in a public access directory then other groups can access it without the knowledge of the group that put it there. The group ends up being in the situation of not knowing who's using their code at all. As one developer explained,

> I'm creating code and I don't know who uses it, I know a few people cause they complain when its broken, but there are always more like if I send out a mail message saying well we're obsoleteing this product we're not going to produce anymore then people come out of the woodwork you know I get totally shocked at the number of developers that over time you know just sort of this has infiltrated into other products so um you know a lot of this happens you know is planned architecturally and then some of it you know err people would rather use working code than write their own hopefully cause that's the right way to do things. (8: 2335-2342)

His strategy for managing his end of the dependency consists of sending out e-mail letting people know when radical changes will occur. Often the e-mail needs to be broadcast to the entire company to be sure that everyone who uses the code will be aware of the changes that are about to be made to the tool.

A manager explained that for some of the more common pieces of shared code get managed by a central group:

> for some of the more common shared code like they have some of the shared libraries that are really common and they kind of package those and put a group in charge of them ... So that's how they do it today. Right. We'll try to automate it and we'll still have a central group that manages the database but now their job will be ensuring that the database is accurate and valid and you know there aren't problems with it. (2: 804-813)

Supporting these dependencies by hand is error prone, and so Computer Corp. has an initiative to support these dependencies using technologies, that include Tool Corp.'s product. However,

these kinds of dependencies cannot be supported easily with technology, as a number of people at Computer Corp. explained to me with respect to the adoption of Tool Corp.'s product:

> ... it's unrealistic to expect 700 developers and billions of lines of code to go into one database so we have a inter-database problem ... It's huge problem. (2: 772-776)

Tool Corp.'s product relies on a database to store all the system components. The developers at Tool Corp. used their tool, and its database to store their code. In fact they used two instantiations of the tool, and two databases, to store their entire product. If the sub-system was not in one database it was in the other. I did not witness or hear of any problems sharing code, because they had a relatively limited search to find any code that they happened to want.

At Computer Corp. the scale of the problem was significantly larger, as one manager explained:

> The issue we have with it that inside the database it's very good at that. Our problem is beyond the scope of a single database. We're going to need upwards of ten possibly so when you and we don't have visibility across databases as we speak today. So one of the projects in place is to get a database to database communication going that has more of a two faced database transaction kind of communication. (5: 1296-1300)

Computer Corp. had also run into problems associated with creating that type of technical solution for managing shared-code dependencies. In the tool, the code gets stored along with the name of the developer working on it. However, at the level of inter-group shared-code dependencies the name of an individual developer turns out to be not so useful:

> There's no abstraction within the tool for a group, there's no name for a groups, you can't group people together and give it a name, you can't have things owned by a group, um and all of that is not too serious when you're talking about it as groupware but when you're talking about it on a corporate basis now you're looking at a higher level you don't really care about individuals anymore you know the corporation doesn't see Joe who's working on this product, they see the product, right, and they want to deal with it at the product level not at Joe's level. So when they see something show up in the central corporate repository they don't want to see stuff owned by Joe they want to see it be owned by the product which is the group right. (3: 840-848)

Having individual names associated with sections of code may provide some information for others; for example, a name to go to when the code breaks. At the same time, it makes finding other pieces of code difficult. Often the developers can identify the purpose of the software when they know which group developed it. This often occurred in the conversations that I had with developers where they talked about other groups' code by referring to an acronym for the group itself rather than describing the code or naming individuals.

Shared code dependencies exist between different sub-systems. In their technical form they create the desired functions of the product; sub-systems interact with each other to provide the desired functionalities and outputs. At the same time they create dependencies between the different groups working on building those relationships between the code.

5.4 Inter-organizational Dependencies

Like Tool Corp., and Contract Corp., Computer Corp. does not develop software in a vacuum. As they move forward on their development trajectories for their current release as well as their future directions as a company they must shift and adapt their plans to accommodate changes in their market. In this section I describe some of the dependencies that Computer Corp. must manage as a commercial software vendor in the high-performance real-time systems market.

Vendor Dependencies

Computer Corp. depends on vendors in two ways. They depend on the vendor of the tools that they use in their development environment. As my introduction to the company came through a Vice President at Tool Corp. I learned particularly about the dependency between Computer Corp. and Tool Corp. However, Computer Corp. also depends on other vendors with whom they compete for their business. Developers and managers at Computer Corp. spoke about these kinds of dependencies also, and their affects on software development itself.

Computer Corp. chose to bring Tool Corp.'s tool into their development environment to help them address their configuration management concerns. The highest levels of management had discovered that the company could not always guarantee that they had a complete version of the product that they had released in source code form. This led the company to review the ways that they currently managed their development process, and the role of tools such as RCS and particularly the home-grown tool, Alpha. As one manager explained the decision came down to a buy versus build decision, between Tool Corp.'s product and Alpha:

> And so we needed to have um well we felt it would be a more um state-of-the-art tool, it turned out [Alpha] was as about as state-of-the-art as you could get but we didn't want to continue to pour resources into it because a business decision if you're going to do that is to productize and since we weren't going to productize [Alpha] we'd better go buy a product, and so that's the decision we made. (5: 1129-1133)

They opted for bringing Tool Corp.'s product into their development environment as they saw advantages in doing that. First, as the manager explained Alpha consumed resources in the form of developers assigned to maintain Alpha, even though the tool would not be sold and only used in-house. Second, by bringing in a vendor, Computer Corp. could benefit from upgrades:

> Because we're going towards in this company in this company towards which is a third party tool instead of providing our own tools here within [Computer Corp.].

> So that the changes we will keep a breast of changes by um by the third party, by our you know tool. (6: 1483-1485)

At the same time that Computer Corp. gains benefits from Tool Corp. it also loses control over the implementation of certain features. I have already described the directions meetings that Tool Corp. holds to solicit comments about future directions from preferred customers. Computer Corp. is one of these preferred customers and they bring their requests for changes and improvements to the product, to the meetings:

> Actually what we're doing is lobbying the to get changes made to future releases. It's the best we can do, but, it's one of the advantages of being a [preferred customer] company. (3: 848-849)

However, unlike Alpha, which Computer Corp. could develop and customize at whim, Tool Corp. has its own agenda and sometimes they do not agree. Computer Corp. remains dependent on Tool Corp. to maintain and upgrade their product. Computer Corp. hopes that Tool Corp. will continue to follow a trajectory of development that suits their own development needs.

The developers at Computer Corp. must also manage relationships with vendors in their own development. Computer Corp. attempts to build "open" systems, and so it must develop systems that work with other popular software. At Tool Corp. I described how this impacted the directions of development broadly. While at Computer Corp. I found developers who must work with the consequences of other vendors revising their products routinely.

I spoke to one developer was hired into the organization particularly for his experiences with a software package that Computer Corp. wanted to integrate their software with. His own development, the integration between the two products, was routinely impacted by changes made to the vendor code, as he explained:

> what we did was when I first implemented the code when I ported it, I checked that version in so that's how you got it from the vendor, and you may want to see for instance if you're port this new code that they have for you need to compare that code to find out how it's different from the original base to see what you're put, what port, what implementation that you have right now and put it in there. Right, it's a way of learning about what changed, right because the thing is doesn't not have our changes, basically you have to take their base and compare it to what they've given you now and see the differences there and then you have the choice of either implementing it from their new code or putting their changes into your new code the code that has the changes. (6: 2021-2029)

These vendors change their code, not only in terms of functionality of individual modules, but at higher levels, in the architecture of the overall product, over time. But, other software development organizations who depend on these vendors, pay a high price for that dependency, because each time the code changes, they must adjust their code to match.

The developer quoted above spends much of his time reworking his code to accommodate the new changes to the integration. Although he may hear rumors about potential changes coming, he must often adapt his entire development schedule simply because a vendor produces a new upgrade, or worse yet, an entirely new release.[23] He depends on the actions of other developers in other organizations routinely, because his code depends on the software developed by the other company.

Customer Dependencies

Computer Corp. manages very direct and more indirect customer dependencies like Tool Corp. Computer Corp. contracts for certain customers, such as government agencies and large multinational corporations, as well as selling their product on the open market. Customers who buy Computer Corp.'s product will expect support for a certain time after they buy the software. Organizations that contract with Computer Corp. may demand that they guarantee support for a certain length of time after the customized system is delivered.

Either way, Computer Corp. must support multiple versions of the software simultaneously. Simply put, the customers depend on Computer Corp. to provide this support,

> We have to support multiple versions of the software simultaneously, we have to support um both in time you know things were released sequentially but customers have contracts where they say you know within in reason they're not obligated to move up to the latest version of software, and yet we have to keep delivering bug fixes of certain severity. Eventually we can determine, we can declare an operating system obsolete, where a customer can keep running it if they want but they can not expect us to continue officially supporting it make sure that things are backward compatible and that sort of thing. But it doesn't happen instantly, you know it's not when we release new OS it's not like all previous ones are immediately frozen so um if I'm a developer out in software development land I may have several variants of my products that are targeted to serve the next up coming release of software. (9: 2563-2580)

Computer Corp. may wish to innovate by design and develop completely new products. However, at the same time, customers depend on Computer Corp. A tension exists between being innovative and supporting existing customers, which is one aspect of the complex customer dependency that exists.

As well as formally expecting support for products, customers influence the directions that Computer Corp. takes in their own development. These influences are less direct, and to my knowledge Computer Corp. does not have directions meetings with preferred customers like

[23] In this respect Cusumano and Selby studied a very a-typical organization when they visited Microsoft. While many other organizations may have to adapt and fit their code to Microsoft's latest operating system, database, word processor or spreadsheet, the reverse does not seem so likely. Microsoft's extremely privileged position within the market seems to suggest that they feel the burden of dependencies much less than other organizations like Computer Corp.

Tool Corp. Customers exert these influences in two ways; what they want, and when they want it.

A project leader pointed me towards the influences that customers have on determining what features should be in Computer Corp.'s product. At the same time he also highlighted the implications that these demands have on the development of the product over time:

> Well that can be one of my jobs, and has been so um one of the things I've been trying to do is to kind of re-establish respect and um credibility um on the part of one group with another to really educate the other group about um the market forces and competitive forces that are driving the first group to change the requirements that um leads to what we need to do then... Right, and that can be hard, laughs. I have history with one of the groups. Yeah and at my level and in my role that's what's expected of me is to sort of um I'm expected to be um a [Computer Corp.] advocate not a group X or group Y advocate. (9: 2802-2808)

In this case the two groups (X and Y) had diverged over the years due to external influences. However at the same time they had code dependencies, their software had to work together. This had led to conflict and tension between the groups, as a result of these external pressures pulling the groups in different directions. The project leader for these groups was caught in the situation of having to smooth over these differences created by market demands.

As well as influencing what gets put into the latest versions of the product, customers create a demand for new releases. However, this is not strictly a customer dependency, because other vendors releasing rival products also force Computer Corp. to speed up their life cycle. However, behind the need to compete is a market that has not been saturated for the products that Computer Corp. and its rivals produce. This demand has begun to put a huge stress on Computer Corp., as a senior manager described:

> Open means release every 18 months, 15 months, 10-12 months its shrinking and shrinking and shrinking. There's more complexity in the world so to get all that straightened you need 30 people delimited is one way to do it but when the overhead component of it is this cost insuring so those thirty people have got to, maybe they'll actually shrink an absolute number but they've got to shrink in the sense that their productivity grows, they have to be much more effective you know they have to just do their thing quickly because they don't have, I mean life cycle pretty much was five years ten years ago now its down to like less than two years and its shrinking, we got get to release in some record amount of time in a year from start to finish a whole new turn everything and everyone's looking around at me in a kind in a daze like I don't want to spend another year like that again you know it was a very difficult task. So, you have to somehow learn how to do this as a business as a routine not as a yet another heroic event. (5: 1458-1468)

Customers may directly depend on Computer Corp. for enhancements, features and support. At the same time Computer Corp. must follow the demands of the market to ensure that their

product remains competitive. This customer dependency plays out in the features of the product itself, and although it may appear that the customer could fall victim to choices made by Computer Corp. about what to provide, if Computer Corp. does not supply a certain set of those features then the customer may choose to purchase new systems from a rival.

5.5 Summary

Research at Computer Corp. reveals that the organization shares many of the same dependencies as Tool Corp. They must manage parallel development, integration, expertise, and change dependencies. However, the scale of operations, the size of the product, the number of developers involved, and the complexity of the code, means that Computer Corp. has developed strategies for coping with these dependencies that are embodied in the organization itself. Unlike Tool Corp. Computer Corp. needs to employ special people, "builders," have organizational divisions like the release area group, to manage these dependencies. That may not be the purpose of the builders or the release area, they have technical definitions for their jobs, but in reality they end up working on the social aspects of dependency management.

Computer Corp. has a distributed development environment unlike Tool Corp. The main development operation consists of several large buildings and they also have developers working in different states and around the world. All of these development efforts must remain aligned so that the product fits together. Currently formal procedures, managerial strategies, departments, committees, and informal communications networks work well enough so that the company can release software. However, Computer Corp. recognizes that these methods are susceptible to failure, and has begun to pursue other courses of action.

Finally, just like Tool Corp. Computer Corp. finds itself in a software development world. Other vendors as well as customers influence the directions that Computer Corp. takes in its software development efforts. However, customers and other vendors alike also depend on what Computer Corp. does to set their own trajectories. Software development worlds have a critical balance of influences, so together companies move forward in competition, but also, necessarily, as allies.

Chapter 6

Case 3: Military Contractor Adapts to Policies

> What type of tools do you need, at that point we've got to start looking at the future of the organization, where's it going, what's its life expectancy, a lot of organizations have only one mission in life. You know if you're at some location working for the [DoD Agency] you have an organization that's only responsible for maintaining one electronic warfare pod on a plane, well that may have a 20 year life cycle, you're into the tenth year so you're half way through, and they're still using VAX's they're still using FORTRAN compilers, and a whole lot's not going to change, so you have to think of why give them expensive tools to just make a minor improvement that they're going to throw away in ten years. (1: 649-656)

This chapter covers Contract Corp. a small military contracting company. I review inter-organizational, organizational, and individual dependencies that impact the software development process. The emphasis is on highlighting both the similarities and differences between a military environment and a commercial one.

6.1 Contract Corp.

Contract Corp. was founded in the late 1970's as a contracting company. Today they compete for both governmental and commercial contracts to supply customer specific hardware and software systems. They have specialized in providing application specific systems for tracking, control centers, and large data handling systems.

The company has a number of locations in the United States and around the world. Despite being a publicly held company, it remains difficult to find out exact details about the company. The division of the company that I studied concerned itself with government contracts exclusively, ranging from unclassified projects for agencies such as the Federal Aviation Administration to national security, black environments.

As well as writing systems for customers the company also engaged in a small amount of internal software development. This would happen when the organization needed a certain kind of tool in order to complete a contract requirement and they could not find one available commercially. So, in addition to having configuration management demands for their contracts, they also used a manual procedure in-house for these home-grown tools.

As a contracting organization the size and nature of the company changes dramatically as the contracts come and go. At the time of this study the company was between two large contracts

and there were few software developers present. This coupled with the sensitive nature of working on government contracts meant that the data from this site was limited and consists of two of interviews with a senior project manager and a configuration manager. I used semi-structured interviewing protocols and conducted over 3 hours of interviews with them. The access I obtained influenced the kinds of dependencies that I was able to find, with most of them concerned the social world of military contracting software development. As a consequence the order of the sections is reversed in the chapter, beginning with an analysis of inter-organizational dependencies.

Despite these limitations the data raises some important new insights and extends and adds to the two previous chapters. Despite the limitations of the data gathering, military contracting environments provide an important contrast class for the previous two chapters. Large scale software development also begun in the military and government contexts, and many configuration management text books acknowledge this heritage — some focus exclusively on military software development standards — so I believe that the opportunity to revisit configuration management in this context proves worthwhile.

6.2 Inter-organizational Dependencies in Military Software Development

This section describes the inter-organizational dependencies Contract Corp. manages. These dependencies differ from the kinds that Tool Corp. and Contract Corp. manage because of the military software development context. However, they do share common properties that are described in this section.

Vendor Dependencies

At the same time that the contractors may have limited relationships with each other, contractors also end up being dependent on vendors. In the past, military systems may have been free of commercial products but not today. Commercial Off-The-Shelf (COTS) development comprises a large part of military development. Sometimes military applications center on a COTS product, and then the in-house development involves building security, process, and other military specific wrappers around the product. Other times contractors wish to incorporate a certain functionality into their system, and simply build an integration to a COTS product. Finally, contractors may choose to use COTS products in their own development environment to help with the project. Often military contracts demand that tools and platforms used in the development process become part of the deliverables — this is discussed in more in the section on classified environments — as well.

Deciding to use COTS products proves difficult, especially in large projects because of the length of time that the system must be operable.

> Most of these very large expensive systems require a life time, a life expectancy of twenty years, so we have to ensure our customer that yes everything we build can be supported for 20 years. The government pays a lot of money for these

> support contracts and we have to make the buy decision, can we buy this service from the general vendor, or do we develop something like the [ContractX] and support it ourselves. (1: 467-472)

However, COTS products do get used, often when the expense of building in-house proves too great.

The decision to use COTS products though, creates a new set of dependencies, between the contractor, the vendor, and the customer of the software being built by the contractor. Instead of controlling the functionality of the product, they depend on the vendor to provide the appropriate behavior. However, military systems have certain safety requirements that make relying on commercial products very dangerous, financially, and in terms of lives.

> If you go from one version of the compiler to the next usually there's new switches, typical thing is optimizing, your code is optimized, runs a little faster, just because things are organized a little differently. If you're an embedded system, ... warheads, guidance systems, or launch trajectories, tanks, you need to have the exact same binary. Right down to the bit, no changes. That's the question, can you generate another binary image, and the general case is the answers always no, because tools have changed, CM systems have changed, binary files have change, binaries have changed, objects have changed, your source code has changed, and nobody knows where the original combination is. Is it important to have that particular binary image, if it is, we have to have a series of tools that will ensure that everything that developed the original binary can still develop it today. (1: 711-721)

Far sighted developers and project managers can predict that tools will change between versions, and so the same binary will not be produced. Therefore the system will not execute in quite the same way. However, other hidden properties of vendor products can impact development, as the project manager also explained:

> Far searching compiler business will know about future improvements so the compiler can automatically take care of things, or take advantage of things that don't exist yet. So when you make a hardware change also the compiler is now looking better because the engineer knew that you know we're now doing to have a 40 point processor that's different, so I'll detect when that processor is there and do it different. So just having different hardware can sometimes affect your binaries coming out the end. (1: 728-734)

Commercial vendors have their own priorities and demands. Because Contract Corp. has chosen to use COTS products, they now find themselves dependent on these marketing decisions. These dependencies manifest themselves as a need to maintain control over their environment, either by putting all the tools under configuration management, as well as knowing about all the hidden affects of changing one component in their development environment.

I have described the affects of hidden functionality on the development environment at Contract Corp. However, Contract Corp. also depends on the vendor delivering the functionality that they promised in their sales pitch. This does not always happen:

> Anyway, we took the product and looked at what was underlying it and said OK, it looks like it'll work and we bought it. This was back in '90. We went away for three months six months and we did requirements analysis and other things came back and took a look at product and said OK now lets make it work sat down couldn't make it work. (1: 399-402)

This left Contract Corp. with a dilemma about how to proceed. Whether they use the COTS product in their development environment, or bundle it into the system that they provide to the customer, Contract Corp. has to cope with having been sold "vapor ware." In this case they solved it but at tremendous personal cost to themselves.

> we made a conscious decision to either throw it away go with another product throw it away and develop another product buy another product, or fix the problem. And we looked at all those and we said we're going to fix it. It's a make or buy decision. Three years later we fixed it. (1: 408-410)

When the COTS product becomes part of the solution for the customer, then the contractor creates a dependency between the vendor and the customer as well. If the customer decides to follow a path of upgrades then they instantly have to assume new costs, beyond the cost of buying the upgrades themselves. As the project manager explained,

> they [vendors] have a tendency to bring out new products, this is good, but they're not always 100% applicable and applicable means more than just the software. Sure the old stuff still works and that's good, don't ever change that, but, what new training is required, you've changed the interfaces, you changed the documentation, we have to retrain. So we have to consider how often they do that, if it's aggressive company, nine months, every nine months they ship a new modification, over ten years that's 12 that's 13 retrain, that you have to go through, and that all has to be costed as expense (1: 681-688)

Contractors find themselves both at the receiving end and in the middle of dependency relationships with vendors. They rely on vendors' products in their own development work. When they work they must remain vigilant to the fact that as the products change that impacts their development environment. They must either choose to settle for one version and retain a stable working environment, or upgrade and then amend all the code affected by the product changes. The hardest COTS product dependencies that Contract Corp. faces are those where a change in one aspect of the environment invokes a hidden change in something else. Sometimes Contract Corp. does not even get to this point, because the product that they bought does not do what they thought it did. Then they must either buy another product or fix the broken one and both solutions involve incurring expense that they may not be able to bill to the final customer. At the same time that Contract Corp. enters into a dependency relationship with a vendor, they may also bring the customer into that relationship. When the customer ends up with the tool in

their solution, then they must either settle for that version of the product, or incur costs and delays during upgrades.

Customer Dependencies

I have already talked about customers in the previous section. The most obvious dependency happens between the customer and the contracting organization. In this section, and the following ones, I describe the impacts that the customer, particularly the DoD customer has in software development.

Standards

Standards influence the way software gets developed in very obvious ways. They dictate the exact format of the solution. The configuration manager described this for her own work in the following way:

> The government gives you a base handout, they're automatic, pretty much things that are done on a computer by somebody cutting and pasting. They must have a standard set of what they call their contract data requirements, cdr form. And then they tailor it to fit whatever contract it's going out on so for example we have a version description document which describes the software, it actually has the software files in it, and what version they are, so it's like a it's a document that the government can go back hopefully and pick up and say OK this is what this is. That's, they're all different, they've got some that describe the hardware, and some that describe the software, and some that describe what you're going to do with the software hardware. Some get very technical. (2: 1121-1130)

The government particularly supplies sets of standards so that it need not be so reliant on the contracting agency to make the final solution work. Theoretically, when the agency assembles the working product, it should have all the information necessary to both operate and maintain it. These standards come on top of other practices that also mandate certain procedures, as the project manager pointed out:

> And there's a whole range of subjects in there, TQM, total quality management, configuration management, software quality assurance, a whole range of what we call soft sciences, and then the government adds a whole series of layers on top of that it makes it even worse. (1: 22-25)

> Um, unless you're a large organization like the government that mandates you shall do this, you shall do this, you shall do that, um soft sciences are almost totally unknown. (1: 67-69)

However, while governmental agencies demand quality in their software systems, and try to use standards to enforce that, other organizations interested in contracting out software are usually more concerned with dollar and time demands. The character of software development fluctuates from contract to contract. Sometimes standards create extra layers of work for

Contract Corp. so that actual development time becomes a small part of the total time and cost taken to provide a system. On other contracts development is a large part of the time spent by Contract Corp. In both cases Contract Corp. does not make that decision. Although they may choose to follow certain practices out of their own professional pride, the final decision is made as a function of cost and time to meet the standards imposed on them.

Even when standards get imposed on Contract Corp. developers may have problems implementing them,

> We also have the problem of Ada, introduced about the same time frame Ada is a methodology all unto itself, so if 2167 is a methodology that doesn't correspond with Ada then the two are going to conflict which one should you use? Well, Ada enthusiasts treat it like a religion, you know, thou shall not disgrace Ada, C programmers are the same nature and 2167 enthusiasts are the same. So we get into a lot of religious arguments as to which ones right and it's just like the program manuals out and thou shalt and they thump on their standards just like a southern Baptist with his bible. It's hilarious just sit down and watch that go on for days and days. (1: 300-308)

Standards, like Ada and DOD-STD 2167 may conflict under certain circumstances. At that point, decisions need to be taken about how to proceed. At Contract Corp., and likely most other organizations, enthusiasts for both standards want to continue the "right" way, so the manager lets developers arbitrate.

Requirements

All customers, governmental agencies and commercial organizations have requirements for the contracts that they have up for tender. Sometimes the customer has unconventional requirements:

> If the DoD Agency wants a new communication box, to plug into a Hardware Platform developed in the sixties, we can do that for them. And we will. And their requirements will dictate what kind of CM we use. (1: 549-552)

In this case Contract Corp. had to either get this specific and obscure hardware platform from the customer, or own it. They also needed to find employees with the skills to develop systems on that particular platform:

> Since we do the development here one of the things that we have to make sure that the tools that's we're using match the engineers. Can we change the tool or do we have to change the engineers, usually we have one or the other, we don't get both, in some cases the customer says this is your tool, in other cases, go ahead use whatever tools you want, I wish we had that more often but we don't. (1: 660-664)

At Contract Corp. the customer usually dictates the technical requirements for the project, the tools and platforms to use. At the same time, they end up dictating Contract Corp.'s hiring strategy for that particular project. As a result of this, Contract Corp. has adopted a fluid hiring policy, where people often get employed for specific projects and then leave unless they can be useful on the next contract.

The Buck Stops Here! Dollar Dependencies

Dollars also influence Contract Corp.'s approach to development. Money shapes the development in a variety of ways though, some of which I have already discussed in the context of other dependencies. Companies like Contract Corp. usually bid for contracts that they want. In their bid, Contract Corp., needs to demonstrate that they can provide the solution for a price that the customer can actually afford. Although this sounds easy, in practice the customer does not often provide information about exactly how much money they have to spend on the contract. Instead, managers at Contract Corp. have to guess:

> If you're writing software for someone else you have to look in their wallet. These people won't let you look in their wallets you have to take a good guess, and with competition the way it is you short cut everyone you can to win the job. (1: 64-67)

When Contract Corp. puts a contract bid together, money does not solely determine what they propose to provide in the solution. The project manager described a spectrum of concerns based on safety.

> At the highest level we have what is called the end safety and at the lowest level we have dollars. There's a spectrum in between of all kinds of software typically commercial systems are closer to the dollar impact, if the software fails we find a solution, what's the impact, generally something that affects the bottom line. DoD weapons research nuclear systems space shuttle avionics, airplanes usually deal with someone dying. OK. (1: 4-9)

Contractors like Contract Corp. know that when a customer needs a safety critical system they will pay to have certain safety enhancing features in the final solution.

Multiple Contractors / Contracts Dependencies

Military command and control systems often contain millions of lines of code, usually embedded into military specific hardware platforms. Usually these contracts require that multiple organizations work together to produce the final system, and Contract Corp. finds itself often working with a variety of other companies, large and small. As the configuration manager put it,

> Yeah, the military, because it is a military contract has a problem in that they're dealing with, if you look their lot configuration it's so very complex because of its millions and millions of parts built by thousands of different people (2: 833-835)

> I don't really have any ideas [how many developers and lines of code] because um as for developers I think that at this company we got up to fifteen developers and they were working all on different parts and plus the subcontractors had at least, probably one company an equal amount and the others had four or five. (2: 846-849)

n the contracting environment, limited information passes between the different organizations working on the same system. Although one or a few contractors may win the over all contract, they usually end up subcontracting parts of the project out themselves. This creates a hierarchy of development, which contains contractors and then subcontractors. At this level of stratification the individual organizations can only guess about the participants on the contract, as the configuration manager does in the quote above.

The center of coordination for the entire project becomes the governmental agency. As the configuration manager explains for her work she interacted with the governmental official in charge of configuration management, who was responsible for pulling together all the pieces of the system, being developed in different organizations.

> I talked to people on the government side, I talked to their configuration management in particular. He had been their CM person for 20 years and he had actually been a quality assurance before CM and so his, I got most of what I know from him, and talking to him because I mean he was telling me what the government wanted, and that's basically what we were trying to do, is give the government what they wanted. (2: 1206-1211)

In this situation the development done by Contract Corp. was dependent on the demands of the governmental agency. However, the central coordinator was not stable during the project. Large systems development often takes several years to complete and during that time the people at the contracting organizations and the government change. In the case of this particular contract,

> They, well actually the project managers, this project has had six project managers, since I've been here and that's on our side, no that was on the government side it's had four project managers on our side, and I don't know how many of the subcontractors, but they changed. (866-869)

This contract exemplifies the problems of working in a military contracting environment. The numerous contractors and sub-contractors depend on the government agency, the customer, to coordinate the individual efforts into the overall system. The governmental agency divides the system into different contracts that hopefully reduces the need for different contractors to work together to produce a working system. However over time people join and leave the project that increases the complexities of coordinating the contract.

Although I have described the scenario of multiple organization contracting, I have not explained what affects this has on the systems produced. The social arrangements between these organizations, or the lack of connection, creates unusual systems, as the project manager describes:

> We have [parts] coming in from the field that have lets say a lot, I can't tell you all the languages, but they have Jovial developed on the VAX system, they have PC Intel chips inside of it so we've got 8088X micro code in there, we've got TI chips, we've got some Ada code sitting on top of that, we've got FORTRAN code sitting in there. Um the [part] is just a jumble of components and a lot of these components come from different contractors each one developed separately, so we have dozens of languages. (1: 380-386)

When Contract Corp. got involved with this particular contract, an especially large project, they found themselves working in a heterogeneous environment. Commercial software development organizations have to be heterogeneous in terms of platforms, software and networking protocols, because the market demands it. In order to compete in any market most organizations need to provide their product for a variety of platforms and software technologies, so they have to have the appropriate technologies running in-house, to develop and test their hardware and software specific sections of code. Military contracting environments find themselves in the position of being heterogeneous because previous development contracts have produced very diverse environments.

Contracting environments also become heterogeneous because of the variety of contracts being worked on at any one given time. In either case the organization has to cope with this heterogeneity, either bought on by the contract or the necessity to have multiple contracts. Contract Corp. copes by hiring and assigning engineers who can transition between these different languages and platforms. As the project manager explained to me,

> Another thing we do is we hire people who are adaptable, we like individuals who can change directions immediately, we have a number of people they work different projects in a given day, that's very difficult. In one instance I'll be using vi and UNIX working on Ada code and a couple hours later I'll be sitting with MS Word putting a document together because I've just used edlin on the PC to put some Pascal code together. (1: 597-602)

> And if you're doing something like the [ContractX] that was very difficult because we had to have people who spoke Jovial, FORTRAN, COBOL, and other applications, and some LISP. All in the same group of requirements for the set of engineers, so we had four engineers who could do that and they were moving around between all those languages and environments. (1: 606-610)

When multiple contractors work on the same system all of the participants are affected by the presence of the others in two ways. First, it creates a situation where the agency in charge of administering the contract becomes responsible for ensuring that the division of work minimizes inter-organizational dependencies. Second, both multiple contracts and multiple contractors have the effect of creating heterogeneous development environments. In the first case, developers need to cope with these environments when they either join the initial contract after the decision to build on different hardware and software platforms has been made, or as is

increasingly common in all development contexts, they have to start with existing systems and extend or modify existing code and these existing systems have been built by several contractors.

These interviews revealed Contract Corp.'s strategy for coping with these impacts. They relied people with two critical skill sets. Their preferred developers had working knowledge of a variety of languages and hardware. Second, Contract Corp. also wanted people who could move around between projects, software and hardware easily. Although the project manager recognized the costs of moving people abruptly between these different environments he sometimes had no choice.

The challenges of heterogeneous environments surfaced in Contract Corp.'s hiring strategies. The company often sought people from small organizations, using the rationale that people in those companies would have to do many different tasks. Prospective employees from small companies would not only be more likely to have knowledge of different systems and hardware, but also be able to change tasks abruptly.

Classified Environments

When a system operates in top-secret situations, the military refers to the environment as black or classified. Classified environments create special demands on contractors like Contract Corp. The special security measures that the military demand to keep the project secret impact Contract Corp.'s operation in a variety of ways. Physical manifestations of this included the rooms that had special blinds over the windows which developers would close during secret development projects. They also maintained a careful log of visitors and visitors' badges displayed the level of security clearance that the person visiting had.

Classified environments create special demands on contractor's development processes as well as the customers operating procedures. Usually classified environments demand that the contractor produce a working product, that will not change during its entire operation.

> what happens if you're in the classified environment where the system doesn't change, you're prohibited from changing it. Everything has to be exactly the same today as it will be twenty years from today. The system does one specific thing and that's all it does and it never does anything else. It's a scary environment because you have to forecast will it meet the need ten years, twenty years from now, they call, generally those are black environments or classified environments so we have a version of product that is in a black environment running, and it'll run for the next twenty years. (1: 689-696)

Classified environments in the 1990's may then still use technologies from the 1970's if they have a life expectancy of twenty years.

At the same time when contracting organizations introduce commercial tools as part of the solution for a classified environment, they create a dependency between the customer and the vendor of the COTS product that needs resolving. Specifically the customer needs support for the commercial product, and this creates two problems. First, the vendor of the tool will

probably not be allowed into the classified environment to provide support. Second, the vendor will not guarantee to support a specific version of their tool for twenty years.

Therefore the customer must hire people who's function will be to support the tool during the entire life cycle:

> It's a government operation so they just hire additional government people that'll work there for the next 20 years, and part of our requirements, part of our job is to train their people on how to use the system, maintain it, manage it, and handling all the problems of product. (1: 702-707)

If during this time, the vendor of the COTS product fixes any problems with the version of the tool installed in the classified environment, the customer can not upgrade. They must keep using the old version until the operational phase of the system ends.

Development, Maintenance and Operations

In the discussion of classified environments the relationship between development and maintenance became an issue. The choices made during development create what some economists (like Rosenberg, 1992) call path dependencies. That means, that choices made earlier on affect actions upstream, because commitments have already been made. The project manager at Contract Corp. recognized that both action and inactivity shaped what happened in the future:

> What do you do in development that will impact the maintenance side, the operation side, what don't you do, in development that impacts your operation and in larger systems it's become very typical to include the configuration management and all the other practices of soft science in the operation side so that the tools we use to develop with are actually the same tools that they'll maintain with. (1: 32-37)

I described the extremes of classified environments previously, but all military environments have much longer life-times than vendors like Tool Corp. and Computer Corp. deal with. Even in non-classified environments, contractors like Contract Corp. must plan ahead, and consider what kinds of path dependencies may arise from the choices they make, for several years.

> But you also have to consider who's going to maintain the system very seldom does the developer also become the maintainer so we have to take into account the quality of people who are going to be using the system 10 years from now. Not just next year but 10 years from now. And what's the cost to the user. You know, if we're using a very complex system that requires a lot of specialized administrators a small staff to just maintain the tool itself you're going to increase the cost to the end user, they're not going to be very happy because all of our contracts require that we give them initial bit at the beginning but it includes the entire life cycle. If our solution has a very expensive life cycle cost we'll never

> win a contract, so we have to balance, what's good and what's expensive with what's effective. (1: 668-678)

Clearly the downstream aspects of software systems, maintenance, and operations depend on the decisions taken during development. However, contracting companies can not decide to do everything that they might, like total quality management, software quality assurance and so forth, even if they would like to. At all times they have to balance the potential pitfalls of avoiding upstream activities against the costs of those implementing those procedures, techniques, and tools.

6.3 Individual Dependencies

I learned much less about the dependencies that the developers working in military contracting environments must cope with to develop software, because I did not speak with any programmers. I did learn about some dependencies that developers face with time, and other personnel on the project. In this section I describe those dependencies, and their affects on the software development process.

Historical Dependencies

At both Tool Corp. and Computer Corp. developers routinely go back to previous versions of the code to find out what changes other developers made. At Contract Corp. they have another reason to visit the past, to recreate environments that they must extend and modify. Like all software military systems grow old and become obsolete. Contract Corp. bids for contracts to revise and update these systems, and to do that they must understand how the old ones were generated.

Unlike commercial settings military software can grow very old before it gets replaced. Whereas Computer Corp. and Tool Corp. developers would go back routinely a few months, or unusually three years, Contract Corp. staff sometimes found themselves regenerating environments from twenty years ago. This following quote illustrates the problems of working in so far the past. Contract Corp. needed to regenerate a binary, X', and when they compiled the code, they got X" instead. At first they assumed that X' had been developed by different source code, but:

> turned out, X" and X' were compiled from the same piece of software two different binaries just by recompiling we'd solved the problem. Everyone sat around going huh, and this was, this particular piece of software was very benign, and there wasn't a man-safety issue so it was fine, so we basically had two binaries coming from the same source code, same compilation, same everything except hardware, but they were fifteen years apart. And the new compilation worked and the old one didn't. We went as far as to even resurrect the old hardware and recompile it, still different binary, X" came out. How was X' ever generated, we don't know, but by recompiling it fifteen years later the problem was solved. If it was a crucial component I'm sure we could spend enough money

> to figure out exactly what happened go back and interview all the original engineers and you know maybe X' was patched that's always a possibility. But since it's not a critical component and the problems been fixed, we're just going on with life. (1: 773-785)

In this case the developers had gone to the lengths of recreating the exact development environment to recompile the software, to reduce changes that might have crept in from commercial products. The code was fifteen years old, so this required finding tools and technology from the very early 1980's. It also demanded that the developers have the skills necessary to install and work on these old systems. Although this particular piece of code did not require intensive investigation as to how X' was developed, the project manager clearly suggests that had it needed that they would have had to go and interview the developers working on the project at the time.

This is an extreme example of the difficulties of working with what has been called legacy code, old pieces of the system. At Tool Corp. I showed how developers depend on the work that their colleagues did in the past. Contract Corp. takes this to the extreme because of the circumstances of military contracting situations. Contract Corp. often finds itself working with legacy code when it joins a contract. At the same time they find themselves in the position of guessing how developers in the past solved a problem, how they generated files, and what the software development context looked like. This requires reflecting back to the practices and standards in place at the time, as well as recreating the technical hardware and software environment.

Configuration Management Tool and Practices Dependencies

At Contract Corp. the developers depended on their configuration manager, the embodiment of their tools and practices. The configuration manager took responsibility for ensuring that the developers and managers produced all the required documentation for the project. She depended on both groups to get her own work done, and spent considerable time adjusting her working practices, communicating with, these two groups. In this section I describe how her technical work of producing documents generated dependency relationships.

Military contracting environments following standards like DOD-STD 2167A must produce many reports accompanying the software and hardware that they develop.[24] These documents relate to specific pieces of code in the system; they perform the accounting function within configuration management work. The documents depend on the software because when the software changes the paperwork must be updated. At the same time the configuration manager depends on the developers to tell what has been changed so that she can update the documents. As the configuration manager says,

> When I came on, they told, my first boss said that my whole job was to be a document cop which meant that I was to ensure that the documents were written,

[24] Newer standards such as MIL-STD 498 for software development have made efforts to reduce this paperwork as there is a growing concern among the software engineering community that focusing on documents has reduced the quality of the actual software built.

> that they were written according to a standard whatever military standard and that they were shipped on time. And I was not to concern myself with anything else. So that's pretty much what I did. Later on it became obvious that the documents were related to other parts of the project and so we needed more control over other things. (2: 872- 879)

The documents describe each step in the process. While some reports give high level summaries of what has happened, others require very low-level technical details. To do her job, the configuration manager needed the developers working on specific sections of the project to complete various documents. To know what she needed from developers she devised strategies to find out. Particularly she would ask them, but also surf the network for documents to see where the developers were in the documentation process.

> So that happened, whenever we shipped a tape I would keep the electronic copy and I would maintain that version, and a lot of times I just went around and asked people what they were working on, and you know be sure and give me a copy remember. It's still funny how much, routinely I would, we have a network server so routinely I would get on it and go network surfing and see what people were working on. (2: 1012-1017)

These strategies helped her to align her own efforts with those of the developers. As she relied on them completing detailed descriptions of the software that they had developed to complete her own required work, she had to devise strategies to find out where everyone was in their work.

As well as relying on the developers for the descriptions of the latest changes, she also relied on the project leaders in her document production work. Project leaders for the contract changed a number of times at Contract Corp. while she remained the configuration manager. Each new project leader bought their own preferred ways of working, particularly for her, with respect to document production, that she had to comply with.

> I started out on a Macintosh with just word processing which I personally thought was a horrible way to go, but that's what my first [project leader] wanted. Then I ran into a flat file system filemaker pro to keep the records and I tried to keep track of, I tried to leave some record of what we were doing so that at least I would know if you know a mistake something happened. It was very simple. And then they got DBase4 which I never worked with for UNIX and we tried that for a while and then they decided that was too hard. So then they took me off of the Macintosh and the Sun computers and they gave me a PC because we had a [project leader] again, and he preferred the PC, so, I mean this was all [project leaders]. This was the funny part was that every [project leader] came in and wanted my reports in a format that they were familiar with, so my files kept going through different systems, I ended up on PC using access to keep records, a database, and that's just because that's what my last [project leader] felt comfortable with. (2: 970-983)

These changes in systems impacted the work of the configuration manager because she had to learn new products and devise new ways of producing the required documentation. The technical dependency between her and the project leader involved the production and verification of the DOD-STD 2167A documentation associated with the software developed. At the same time this technical relationship put the configuration manager in the situation of depending on the developers for the information to go into the documentation for the customer, and on the project leader's preferences for document production.

Code Dependencies

The developers at Contract Corp. face the hazards of changing code once they've built the system. As I could not interview developers it was difficult to know exactly what dependencies they had to cope with on a day to day basis; however, clearly they existed because the manager spoke of strategies they used to get around the problems of managing them. Specifically, the manager described hiring and financial incentives they used to encourage developers to remember as much of the code as possible:

> If it's not working, like a configuration problem, it's more than just changing a line of code, there's a ripple effect in the system, and all of a sudden they think, ah I've got to change it here, change it there, they're thinking through the entire process. I've seen them sit for a half hour just thinking through all the different ripples of what's going on. They sit down and read the mental work we pay these people a lot of money because they are very good. (1: 140-146)

> Um, we measure, in this organization we measure the quality of the software engineer by his ability to intimately understand lines of code. We measure that is it a hundred lines of code, is it a thousand lines of code, 10,000, our senior and best engineers intimately track 60,000 lines of code. This doesn't include comments, these are all lines of code, original source code, um. They're hard to come by, a lot of experience. Our typical engineer can at the entry level 500 to 1000, our general working staff can do upwards of 5,000-6,000, it takes um a lot of dedication to go beyond those points, you have to live it. (1: 128-135)

Clearly developers at Contract Corp. have developed two strategies for coping with these ripple affects. First, good developers "live" their code represented by their abilities to track large amounts of code in their minds. Second, beyond that they can visualize the behavior of these sections of the code that they know. They sit and think about the impacts that a change will have on the software that they know. These strategies have one severe limitation; what if the change impacts code that the developer has little or no familiarity with? Evidence from Tool Corp. and Computer Corp. suggests that this may be a problem for Contract Corp. It may also not only affect Contract Corp. but other contractors working on the same overall system, but different, yet dependent, sub-systems.

6.4 Summary

Software development in the military contracting context highlights the complex set of dependencies among contractors, customers, and vendors. Building software systems involves managing these relationships. This case brings demonstrates how complex these inter-organizational dependencies become as they last a long time and involve more organizations than inter-organizational dependencies in product development.

Individual dependencies at Contract Corp. also have unique features. Standards used in military contracting focus greater attention on documentation activities. Much of the work that the configuration manager described was a process of aligning the documentation with the software.

Chapter 7

Dependencies in Software Result from: Systems Change, External Influences, Multiple Products and Integration

> What you really have is um complex dependencies so it's a layered dependency problem so it's not like every dependency is exposed in the first order to every other space it might be second or third order dependencies out there. So in that sense it has this dimensionality to it that makes it feel parallel but it really isn't parallel it's just a dependency tree that's really um weird, really hard to even visualize what it might look like or even, we're thinking of starting an initiative that would be a whole task force or specially chartered group to examine dependencies. Just because you know that is such a hard problem for people because it bites, it's enterprise wide dependencies, right so how do you manage them, well right now we don't manage it, we stumble over it, and try to solve it every dependency one at a time, and so I, this we use group is like critical to us. It's solving that dependency issue. Configuration Manager, Computer Corp.

The previous three chapters presented data gathered from different software development organizations. Each of the chapters focused on one organization and examined the dependencies that the developers, groups and the company face in their work. This chapter groups those dependencies by the events responsible for them. These are: changes made to the system, external influences on development, the necessity of building multiple products simultaneously, and the need to integrate the software components. Before focusing on the causes of these dependencies, four differences among the three sites are discussed. These differences influenced the way that each organization coped with their dependencies.

7.1 General Observations

Four aspects of the organizations studied form a useful foundation for understanding how these organizations differ in their strategies for coping with these dependencies: the effect of the size of the organization on the approaches to coordinating software development, the challenges of heterogeneous environments, length of time using any technological support for developing software, and differences between contract and product environments.

Size of the Organization and Scalable Solutions

Software engineers know that the size of the development effort makes a difference in terms of the implementation of development methodologies. In their discussions of scalability they frequently ask whether a technique from programming in the small works for programming in the large (Tichy, 1992). Software engineers focus on the technical aspects of that problem, for example: does the notation produce unwieldy specifications for complex systems, does this testing strategy work in very large cases, and does this method remain workable for big sections of code?

My study suggests that another critical dimension of scalability concerns the management of software development. In this study I described two organizations that were very different in terms of the size of their development effort, people and code. This was reflected in the actions that each organization needed to undertake to coordinate their software development effort.

What Tool Corp. can achieve with 14 developers, one part-time build manager, and a project manager, takes Computer Corp. developers, people employed full-time as builders, steering committees, and entire departments to accomplish.

The nature of the solutions that Tool Corp. and Computer Corp. have adopted to manage these problems differs as well. Both organizations use informal procedures, e-mail, and face-to-face communication to manage some of their dependencies. Both organizations also have some organizational functions to help manage other dependencies, such as project managers, and having someone do the build. However, the size of the development effort means that Computer Corp. needs to provide departments, committees and special job functions to coordinate the development efforts across the entire organization. Clearly, scalability of dependency management solutions has to factor into any technical or managerial decisions made.

Heterogeneous Environments and Tool Usage

Another stark difference between Tool Corp. and Computer Corp. concerned the homogeneity of tool usage. At Tool Corp. the development environment contains a variety of machines that all run the product. The equipment exists within the environment because the developers need to build and test versions of their product on those machines. The tool support remains functionally consistent across all the machines: although the interfaces may change, the intent of the tool remains the same. In this respect the consistency of the environment lowers the barriers to providing effective technical support for dependencies. For example Tool Corp.'s developers can make assumptions about the state of development, the presentation of the code, and the builds that happen each night because the penetration of the tool into the environment is complete. Nothing is outside the tool.

The developers also have the advantage of having their entire product inside two databases. The small amount of code — in comparison to the situation at Computer Corp. — means that they have fewer places to look to find other components and that more developers work inside one instantiation of the tool. Developers can manage their dependencies with a greater percentage of the others working on the project through the tool.

This was not the case at Computer Corp. Although the environment consisted of seemingly few platforms, these different platforms supported a number of different configuration management tools. In the absence of a strong push to conform to one tool — the organization was in the process of enforcing consistency — the development groups often varied in the tools that they used to help them manage their dependencies. As such dependency management strategies did not bridge the entire development effort, but remained localized around certain features of the tool that they used.

The number of repositories for code at Computer Corp. was unclear, but in their vision of the controlled development environment the configuration management group estimated that the organization would need at least 20 databases of code. The configuration management group had decided already to implement another layer of technology to support some types of synchronization of software among these databases.

I have compared configuration management systems with groupware systems, in chapter 3 and elsewhere (Grinter, 1995). This research offers a number of observations about tool usage. Critical mass theories (Grudin, 1988; Markus and Connolly, 1990) suggest that adoption of a tool requires that a certain fraction of participants use the technology. At Tool Corp. everyone used the same tool. As everyone used the tool the value of the kinds of coordination that they could achieve increased. Beyond the obvious emphasis that Tool Corp. developers placed on configuration management, two factors made this level of usage possible, the scale of the system itself, and the penetration of the tool into the environment.

Recently Grudin and Palen (1995) examined the use of group scheduling systems in two organizations: Microsoft and Sun. They offered four reasons for the high degree of use: versatile functionality, ease of use, organizational infrastructure, and informal peer pressure. The two organizations that they studied have a common platform standard, PC's at Microsoft and Sun's at Sun. This study suggests that whether an organization has one or more platforms does not matter if the tool works with all of them. Developers working on different platforms at Tool Corp. managed to work with each other because the tool worked on both platforms. However, the developers at Computer Corp. had no common tools that ran on all the platforms they used in their development work.

After Adoption: On-Going Coordination Challenges

As groupware remains a relatively new phenomenon in organizations researchers have often written about the adoption of technologies (Orlikowski, 1992) and the adaptation of existing practices to accommodate the changes that these tools bring about (Bowers, 1994). This study confirms these observations, particularly those of Bowers, in a new domain, software development. At Computer Corp. the developers had begun to adapt their existing practices as they switched over to the new tool.

However, at Tool Corp. they had passed the adoption stage, and perhaps they never had one in the sense that they adopted the tool that they built. The tool helped them to manage some of the dependencies they encountered in their software development work, but failed to support others. I have tried to emphasize how difficult developers and organizations find dependency

management. Dependencies take time to resolve, even temporarily. The tool has mitigated and taken on some of that work, but dependencies don't disappear because they are still in the software itself. Those outside the scope of the current technology still demand other solutions. Some of the dependencies, like parallel development, vacillate between being technologically resolved or requiring manual intervention. In studies of long-term usage of groupware systems we may see these "sometimes" solutions, sometimes taking care of a coordination problem, and sometimes failing.

Different Software Development Contexts

Software development occurs in a number of different contexts. Grudin (1991) identified three contexts of interface design: in-house, product and contract. In this study I have examined two different contexts: product and contract.[25] This study suggests that the contexts share similarities and have differences in the kinds of dependencies that they must manage.

Contracting shares some of the same demands as product development. For example Contract Corp., like Tool Corp. and Computer Corp., must build products for a customer and meet their demands, and the individual developers manage historical interactions between old and new versions of code. However, both these activities have their own peculiarities based on the context of development. In the contracting environment the customer sets many more formal demands on the contractors' development process, which come in the form of standards, financial constraints, and in their most restrictive form classified development. Also, developers may find themselves working with old code, not nine months or a year old, but perhaps a decade old. What makes that different from product development contexts is that this old code may not have been changed in ten years. These differences shape the skills required of the organization and the developers.

7.2 Individual Dependencies

Individuals within the development team often need to coordinate their work with each other. The process of evolving the whole product involves continuously aligning the system components with each other, ensuring that changes get implemented in groups, and revising previous versions of software. Developers engaged in these activities must manage the technical relationships between the modules. Simultaneously developers engage in articulation work by negotiating and coordinating their efforts with others working on those related parts of the system. I called these relationships individual dependencies' to capture the fact that these relationships involve individual developers, those responsible for the related pieces of code.

I have described the social and technical aspects of each individual dependency found during the study. I have also discussed the strategies that the organizations use to cope with these dependencies and the degree of success that tools have in supporting dependency management.

---

[25] Contract Corp. operated in a special kind of contracting environment, a monopsony or single customer environment.

Following grounded theory explanations, I turn to a discussion of the causes of these phenomena (Strauss and Corbin, 1990). In this section the different kinds of dependencies are categorized based on the four conditions that led to them arising: the evolution of the software, the need to make a whole product from the parts, the role of practices and tools in shaping dependencies, and the external demands placed on developers.

Systems Evolution

Chapter 3 began with a quote by Whitgift (1991) about the role of change in configuration management. He claimed that changes create problems for configuration management activities because altering one piece of the system forces another modification somewhere else. He identified a critical part of configuration management work — managing changes as the system evolves — but only talked about its technical implications. As my data shows, the changes made during systems evolution generate a number of dependencies that have both technical and social aspects. In this section I describe the dependencies that occur as the system evolves: parallel development, change, expertise and historical.

Parallel development dependencies begin when two or more developers have to change the same piece of code at the same time. The technical dependency that they have with each other involves the changes that they make. First, each developer must make their own changes to the code. Together the developers must produce a single module that combines all of those individual changes. This technical dependency forces developers to engage in collaborative activities. After they have finished their own changes, one developer must interact with all the others to produce the merged module. The articulation work involved in producing a single module involves learning about the modifications that others made and how those changes interact. One developer typically ends up in charge of the merge. Developers use configuration management tools to help them manage these social aspects of parallel development, but often end up needing to have face-to-face meetings to discuss merging issues.

Change dependencies arise from the concept of a logical change to the system, such as resolving a problem or making an enhancement. This logical change often requires that several components of the system get altered, including various software modules and associated documentation. The translation from logical change into systems components creates the technical aspect of the dependency: components must all be amended to reflect the desired functionality. That work often gets divided among different developers, who must make their own changes and ensure that all their changes "fit" together to fix the problem or add the new function. The developers assigned to the change must either locate their counterparts or the code that the others have worked on align their changes.

At Tool Corp. the developers manage change dependencies by using the configuration management tool that shows them what the latest changes are and who is working on them. At Computer Corp. the tools may help developers manage locally contained changes, the modifications to a sub-system owned by one group. They may also use meetings and other informal communications to help them find out what everyone within their group is working on. However, when changes span groups, developers rely on steering committees to help them align

changes across the organization, and established rules about putting the changed code in publicly accessible places.

Expertise dependencies arise from the relationships between code modules that span ownership. When developers decide to make changes to a piece of code that they know, sometimes they discover that they need to understand something beyond the scope of their working knowledge. When this happens a technical change in their own code relies on a social relationship where an expert informs the other about implications the change may have and the developer designs the solution to account for those possibilities. Expertise, and the reputation for having it in a certain area of the system, also forms a way for managers to divide and assign work. At Tool Corp. developers either know who the experts are or use the tool to find out who has been working on code in that sub-system. At Computer Corp. it gets very hard to keep track of all the people working on related software, although sometimes the developers know which group the person works for.

Historical dependencies arise when developers extend or modify existing code. Technically code evolves as the system grows and the functionality changes. Over time different developers work on the module, because other people change jobs or move into a new area of the project. Each developer that works on revising the module to meet new requirements must learn to understand how that code works to make the necessary revisions. This requires learning not only about the way that the code works, but the reasoning behind it. The more that they know about the context of development, the reasons that the previous developer implemented certain functions, the easier it becomes to pick the best solution to the current problem.

At Tool Corp. people used the organizational memory to help them learn from the past, and coordinate their efforts with the echoes of previous developers. At Computer Corp. the developers also used tools to provide them with information about how things have changed over time, when they had the time. Nevertheless the developers spent more time guessing why the module had developed as it did. Contract Corp. developers also managed historical dependencies, but due to the peculiarities of contracting development contexts they found themselves working with very old versions of the code. The extreme occurs when classified code changes for the first time in ten or twenty years.

Interface dependencies begin with the identification of a problem that needs solving. The person who reports the problem "sees" that it occurs in the interface of the system. Although a bug may appear to be in the interface, the real problem is often inside the system. Interface code depends on the technical substrates below, and the person responsible for the interface depends on the other developers to help find the source of the error. In this study I singled out interface dependencies, but they are a special case of expertise dependencies. The interface developer probably owns more code that interacts with all parts of the system than any other developer, so often find himself going to others to ask them questions about the code.

All of these dependencies arise because the software being developed has a problem or needs enhancing. The technical aspects of all of these dependencies manifest themselves as the relationships among pieces of code involved in the change. Sometimes all the code needs to be adjusted simultaneously, as in the case of change dependencies. At other times only one part of

the code needs amending, but the developers need to understand how it interacts with surrounding modules, as in the case of expertise and interface dependencies. Finally, when parallel development and historical dependencies occur the technical relationship exists between two versions of the same module.

These technical relationships have associated social dependencies. The social aspects of all these dependencies arise because different developers work on the modules involved. When developers enter into a dependency relationship, together with fixing the problem they must spend time aligning their efforts with the development context and others' on-going work. All of this requires learning, either from other developers or through the use of technology. The goal of this learning is to make informed choices about how to create the new revised code, the modules that fix the problem or deliver the required enhancement.

At Tool Corp. the developers learned from both the tool and each other with relative ease. The tool provided considerable information about the evolution of the system components and who owned each piece. As a small team they could easily find one another and discuss potential solutions or align their efforts. The tool also supported on-line fitting, by continually providing other developers with the latest stable changes of the entire system code. At Computer Corp. the scale of the development operation impacted developers' ability to learn about the context of development. Sometimes it even affected their ability to find the latest changes of the code that they needed to align their work with. Having a variety of tools and a highly dispersed development operation they often relied on other groups to take care of these articulation activities, as I shall describe in the section on group level dependencies. Contract Corp.'s developers found themselves facing similar problems with changes. However, their time-scale for development differs significantly from the two commercial environments; the development context that they may have to learn could be ten years old.

Dependencies arise from the fact that as a system evolves it changes. Not all parts change at once, although that may happen. Developers find themselves working with a moving target when they start changing a part of the system. The software that they start fixing does not end up being the system that they have to integrate their fix with. At Tool Corp. the system could evolve in 14 different ways simultaneously, as there were 14 developers there. At Computer Corp. it could evolve approximately 700 different ways. These dependencies as technical and social relationships that need to be maintained during development so that the software components still work together.

Making Whole from Parts

A consequence of decomposing a software system into modules is that the system needs to be built into a whole. Whether or not individual components of the software change they must be integrated and tested to see whether they work together. At Computer Corp. integration happens on two levels: building sub-systems and putting the whole system together. Gathering the sub-system at Computer Corp. approximates the same effort required by Tool Corp. to build their entire product. At both organizations the process of constructing the system often reveals that the parts interrelate in ways that I have termed integration dependencies.

Technically integration dependencies incorporate a number of relationships. Software engineers recognize that modules depend on each other at build-time (an element depends on the components from which it was derived) and compile-time (when modules must be compiled in the correct order). However, when I interviewed and watched developers integrating systems I observed another technical aspect of integration dependencies: aligning current versions of all the modules. This involves gathering all the most recent versions of the components to go into the final system, and ensuring that either all of a change gets in, or stays out.

Both Tool Corp. and Computer Corp. have developers assigned to the role of build manager, the person responsible for collecting the newest changes of the system and putting them together. At Tool Corp. the build managers depend heavily on the tool to help them decide what pieces of code get into the nightly builds, for doing a build, and producing an integrated system. The system tries to take care of much of the work involved in finding the most recent changes and ordering the technical execution of the build.

However, at both organizations a build manager has to manage the social relationships that emerge as a result of the technical aspects of the integration dependency. When a build does not finish, then the manager has to find the code that broke it and get the problem resolved. At Computer Corp. the build manager also has to find all the latest changes to put into their build, which requires considerable effort. It requires intensive interaction with all the developers working on the sub-system to get a sense of what is going on.

Computer Corp. also reveals how as software grows in size the complexity of managing integration dependencies rises. When sub-systems have many components, a developer can not assume the role of build manager, because they do not have enough time to do both jobs. Computer Corp. has a job function, builder, whose job consists of doing nothing but integrating large systems. Computer Corp. also has to integrate the entire system. They have an organizational division, the release group, who perform that function.

Practices and Tools

The three sites also reveal how their configuration management tools and practices not only reflect the dependencies that they must deal with, but in turn shape they ways that they cope with them. At Tool Corp. the tool handles a large percentage of their individual dependencies routinely. For example, it gathers code together, helps them find the sub-system expert, keeps records of how previous developers resolved problems in the code, and provides them with the latest changes. Despite having the tool the developers still need to spend time managing those aspects of the dependencies that the system can not resolve.

The tool shapes the way that they think about these dependencies, and about software development more broadly. The developers at Tool Corp. rely on the tool implicitly and make assumptions about what other people are doing based on information present in the system. However, it was the developers at Computer Corp. who really revealed the extent to which a tool can shape the way that these dependencies are coped with. For example, the shift from Alpha to Tool Corp.'s product was a transformation from thinking about merging entire sub-systems to merging modules. Changes such as these require the developers to completely reconsider what

merging means, both technically and socially. While some strategies remained intact, such as their ability to backtrack using the tool, others changed, and often the developers at Computer Corp. wanted help in making the cognitive shift between the old and new tool ways of working.

At Contract Corp. the configuration manager did all of the work that the tools did in Tool Corp. and Computer Corp. The configuration manager also had the responsibility of making the adjustments as tools changed. The developers remained relatively free from the process of adaptation as different program managers came and went, because what changed was not the development environment tools but the document production systems. The differences created by the configuration management demands of military contracting and the embodiment of those procedures in a person rather than a tool affected how changes over time needed to be managed.

Tools and people support the management of dependencies. At the same time they create their own dependencies, the developers become dependent on the tools and on people providing certain kinds of information and on doing their job in a certain way. Changes in the environment require adaptation of practices and strategies for managing the technical and social aspects of all dependencies.

External Demands

External demands, those coming from outside the organization, also create dependencies for software developers and are discussed extensively in the section on inter-organizational dependencies. However, in one instance — platform dependencies — these influences showed up in individuals' work. Computer Corp. wants to provide its product on several different platforms to increase its market share. However, some of these platforms do not have any configuration management support. Rather than work unsupported, developers work on platforms where the tool exists and cross-compile their code.

Platform dependencies arise when the differences between the two platforms create technical difficulties. In the case that I described the developers had to choose how they worked together as a direct result of the platform differences. Adaptation of practices does not only occur when an organization changes tools. At Computer Corp. it also happened when the organization expanded its product line to compete in new markets. As well as managing the two platforms technically, porting code, running and storing code in two different places, the developers need to adapt their software development practices to compensate for the challenges of multiple platforms.

7.3 Group Dependencies

Teams within the development organization working on the same product also need to align their work. Development groups also need to work together as a group to maintain a shared understanding of the product that they work on. I separated the kinds of dependencies that involve groups in these ways from the ones that individuals manage, because they represent something closer to what Strauss (1988) calls the articulation process. These processes reflect a

need to work with and understand the whole product rather than the pieces. At this level I found two factors that cause dependencies: the necessity to make the whole from the parts and the need to manage multiple wholes simultaneously.

Making the Whole from the Parts

I have already described integration dependencies as the challenge of constructing the whole from the parts. However at the group level I found more dependencies that stemmed from the same need, to create a sense of a whole from the specific parts of the system. Software engineers have begun to recognize the importance of understanding the whole in some abstract way, they call it the software architecture and it has recently become an important research topic (Garlan and Perry, 1994). Again, software architectures emphasize the technical aspects of the problem, designing "better" systems though understanding the conceptual arrangements of the parts.

The developers at both Tool Corp. and Computer Corp. need to work together to generate an understanding about the product. Although the developers spend the majority of their time working on small sections of the software, they need a sense of how the whole system fits together. Having this "big picture" gives them direction and leads them to pick certain solution paths. Knowing the big picture can also help them realize when their software has a run-time interaction with another section of the system, and when they can reuse code from another developer working on a similar problem.

These technical aspects of big picture dependencies require accompanying social support, which was very difficult to provide in both organizations. Establishing this big picture was something that groups within both organizations struggled with. At Tool Corp. the developers could not work together to build the picture of the whole system. They tried to use electronic mail and group meetings to develop this understanding of the product, but it took more time than they had. The tool, while helping them with individual dependencies, had no abstraction available for viewing the system as a whole. At Computer Corp. the developers realized that they would not be able to visualize the system as a whole, but they still wanted to understand their sub-system and its relationships with other parts of the product. Some people, architects, served as boundary spanners for the developers at Computer Corp. by bridging two groups and aligning their development efforts.

Big picture dependencies differ from individual-level integration dependencies. While integration dependencies focus on getting the pieces together in a certain order, big picture dependencies operate at a higher level of abstraction concerned with the ability to visualize the system as a whole. This higher order required the entire team to work together as a group to understand what the system was doing. The need to understand not only how the system fits together, but how it interacts as a whole was especially apparent at Computer Corp. where teams also had to manage shared-code problems.

Technically a product comprises a number of sub-systems that interact to create the desired functionality. This takes teams into shared code dependencies. These dependencies span sub-systems often relying on what Computer Corp. calls shared code: libraries and functions that

many sub-systems rely on. They also require management, and Computer Corp. deals with this by having a central group manage these pieces of code.

Working out how to share the code, aligning efforts across these sub-systems, creates problems for organizations like Computer Corp. It requires setting up places where individuals can access the code, and controlling access to those places so that other people do not change the code and unwittingly cause other dependent parts of the product to fail. These issues have both technical and social aspects, and development organizations must manage both of them to resolve the resultant difficulties.

Managing Multiple Wholes

In any development organization a number of software development life cycles exist. At Tool Corp. during my time there they developed a point release (an upgrade for the existing customer base) and a new product simultaneously. The company also developed each product on multiple platforms, and some of the platform developments took longer than others. The developers were divided into groups working on distinct platforms, and although they wanted to maintain consistent functionality across the platforms they had to separate the life-cycles because they needed to go at different development speeds. Technically, the speed issue manifested itself during the nightly builds. The newer platform code was not ready to be built each night because it would break the build, consequently impacting the work of the developers working on both platforms. The decision to split the code however meant that the developers responsible for developing on the new platform had to coordinate their efforts with the others working on the old platform. This meant that every so often both platforms needed to be compared to see whether they provided consistent functionality.

At Computer Corp. the situation was more complex. The fast development life cycles for the entire product mean that some groups work towards the current release and other groups work towards the next release. The entire product has a life cycle and each sub-system within the product has its own development schedule, too. The organization has to maintain control over these competing life cycles, ensuring that all the groups work towards appropriate schedules.

The idea of multiple life cycles clashes with the visions of software development portrayed in books on the topic. Models of software development appear to preclude the idea that multiple products get developed simultaneously. However, models need to take into account the fact that in commercial environments the days of one product have passed. Companies have multiple versions of their products and many concurrent life cycles, all of which need to be managed.

7.4 Inter-organizational Dependencies

Inter-organizational dependencies situate software development in a complex web of relationships. These dependencies influence and direct the development options of every single development company. The theory of social worlds captures these dependencies because it highlights the affects that other companies have on a software development organization.

Contract Corp. provides the clearest evidence that outside influences affect development. The government influences them in a variety of ways, by setting requirements, standards and financial limits. In certain cases the government also stipulates secrecy regarding development. These are the most explicit conventions within the software development worlds that I studied. However, other conventions exist for all three organizations.

As software development organizations such as Tool Corp. and Computer Corp. embrace open systems they must provide integrations to new products that appear on the market. They cannot control vendor dependencies; usually they must react to them. For example, when a vendor changes a product that the system under development relies on, Tool Corp. or Computer Corp. must revise their development schedule to take account of the new release. Usually this means carrying on with the old version development to capture the segment of the market that will not upgrade, and beginning a new product variant to appeal to customers of the new release. At the same time that Tool Corp. and Computer Corp. depend on other vendors, they also influence what other vendors do by releasing new versions of their products. Recently, some software development organizations have formed partnerships and coalitions with others to try to organize how and when products change so that they can regain control of their own development schedule.

These dependencies may not be news to economists studying technological change, but they certainly do not appear in software engineering literature. The model of software development proposed in project management literature does not question who controls the development schedule. Even authors like Cusumano and Selby (1995) who studied Microsoft did not see, or at least did not report, any affects of other vendors on Microsoft's development schedule.

Customers also influence decisions Tool Corp. and Computer Corp. make. Customers influence the development trajectory of the products by either directly telling the organizations or through the marketplace. Customers may also influence the organizations through consultants, user groups, and more recently the Internet.[26] They also depend on the organizations for support and on-going compatibility. Organizations also use focus groups, where customers provide feedback on the product and marketing analyses, to determine their future directions. Their concern with maintaining their market share encourages them to provide good support and on-going upgrades.

7.5 Understanding Dependency Management

Four causes of dependencies have been identified: the need to integrate the system, the on-going changes, the existence of multiple products, and external demands on the development process. However, these causes have intricate relationships. The connections between the causes are described in this section.

---

[26] Intel was forced to revise their Pentium chip after a mathematician posted the bug to a USENET group. Soon word passed around the Internet that the chip has a problem and that they were not going to fix it. The Internet served as means to complain about Intel's decision and soon the company reversed its decision. The president of Intel, Andrew Groves, also posted a message to the USENET group explaining Intel's decision reversal.

This study initially focused exclusively on the development process as it occurred within organizations, under the incorrect assumption that they had full control over the life-cycle. In addition to impacting the life-cycle, external demands also shape some of the other causes of dependencies. For example, clearly external demands influence the evolution of the system, and create the need for multiple products. However, demands generated within the development organization also affect the evolution of the system, and produce multiple products.

Both the need to make the whole out of the parts and the need to manage multiple products involve integrating the components. As described assembling one product involves physically putting the product together, and also using the understanding of what the whole does to continue working on the parts. Managing multiple products requires keeping those assemblies distinct even though one code module may fit into more than one product.

Finally systems evolution, the on-going changes to the product, implies that integration must occur more than once during the life cycle. Every time a change gets made the system needs recompiling to see whether that change works with the rest of the system and whether it produces the desired outcomes. Systems evolution forces integration to become an on-going activity.

The two single most important causes underlying dependencies are the need to integrate systems and the external demands placed on software development. The latter should come as no surprise to researchers who have studied software development. However, their studies have focused on the early stages of development: requirements analysis and design. This work looks at development and suggests that outside forces influence development. These effects show up as new demands on the system and provide an explanation of why requirements elicitation occurs throughout the development life-cycle rather than being isolated to the few stages. This study contributes to the understanding of how external demands influence software development.

Another contribution of this work has concerns integration. Integration dependencies reveal a fundamental challenge faced by software developers, how to build a system. Unlike systems decomposition, which may only occur once, developers must continually integrate their system whether or not the components change. I call this recomposition, and the final chapter focuses on its significance.

7.6 Summary

In this chapter I have done three things. First, I reviewed and synthesized each specific dependency that I described in the data chapters previously. I illustrated the technical and social aspects of those dependencies as people who work with them understand them. Second, I classified them by their causes. System change, integration and external influences create dependencies between the code modules that form a software system and at the same time among the individuals, groups and organizations that work with them. Third, I described how the underlying causes relate to each other and identified the contributions that this work makes particularly in understanding software recomposition.

I started this thesis work with the ideas about the way that people work together based on the theories of articulation work and social worlds. I have discovered that developers engage in articulation work to manage dependencies. Developers need to work with each other to track changes through the system, build the whole from the parts, control the affects of external influences in their work, manage the multiple product life cycles. They either do this alone, as a group, or with the help of organizational and technological coordination mechanisms, like job functions, departments, and tools. What gets done in the name of configuration management at all three organizations arises from the need to manage these social and technical dependencies. To add further complexity to these dependencies all development organizations find themselves in a social world with conventions that may or may not be clearly articulated. At the same time as they coordinate their internal development efforts they must align their efforts with vendors and customers.

Chapter 8

Conclusions: Contributions, Limitations, and Future Work

> Most software projects are group activities, involving all the complexities of group dynamics, communications networks, and organizational politics. The study of group behavior in software development is in its infancy, but like the study of individuals, it promises to improve our understanding of the development process, particularly at the front end. Many observers believe that improving this phase of development could have the most impact on software quality and productivity. (Basili and Musa, 1991; 94)

This thesis began with a question: how do software dependencies affect the development of systems? The argument that dependencies are technical relationships among code that create and reflect social relationships among individuals, groups, and organizations. This chapter summarizes the line of reasoning in the thesis. It also examines what this thesis has to offer our collective understanding of systems decomposition. Limitations and future work are described.

8.1 Summary of Thesis

Sometimes software fails. One of the reasons why it fails is because it is hard to manage the dependencies between the individual software components. Dependencies are relationships among software and people working on that code. These problems come into focus when you study configuration management activities as they happen in practice. Configuration management involves identifying the components of a software system and tracking how they change over time. It also involves maintaining information about how to assemble the components into systems.

In practice configuration management is the domain of software development practice where managing the dependencies between system components becomes necessary. Configuration management is concerned with building systems from their parts, over and over again, as the system evolves during development. The hardest part of this is managing the dependencies among the software components. The individuals and teams responsible for sections of the code must engage in articulation work to manage these dependencies. At the same time the organization must participate in the social world of software development so that they can continue to integrate their product with others.

I began this thesis with a question: how do software dependencies affect the development of systems? The answer to the question is then: software dependencies affect the development of systems by creating situations where developers and groups must engage in articulation work, and organizations must participate in social worlds.

8.2 Decomposition Implies Recomposition

The argument presented in this thesis clearly suggests that configuration management should be repositioned within practical software development and academic software engineering. When developers, teams, and organizations engage in configuration management activities they find themselves managing complex dependencies. However, the explanation also reveals a more fundamental challenge for researchers interested in software engineering issues. It suggests that decomposition implies recomposition.

Chapter 1 introduced the concept of decomposing systems into modules that exhibited low coupling. Coupling is a measurement of the dependencies among modules. The more dependencies modules have the more highly coupled they are. The solution proposed by software engineers has consisted of designing systems with modules that have as few dependencies as possible.

The underlying philosophy is to decompose systems to eliminate recomposition issues. This is clearly illustrated by the following quote,

> The benefits expected of modular programming are: (1) managerial — development time should be shortened because separate groups would work on each module with little need for communication: (2) product flexibility — it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility — it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood. (Parnas, 1972; 1054)

Parnas clearly believes that good decomposition would eliminate the complexity of recomposition.

The data presented in this thesis challenges this assertion. Clearly the three organizations studied did not have the managerial, product flexibility, or comprehensibility gains that Parnas describes. This raises two questions: did Parnas get this wrong or did the organizations do bad decomposition?

The answer is neither. In proposing criteria for good systems decomposition Parnas identified a set of key issues, that until that time had remained hidden. He provided a compelling argument that trying hard to decompose systems into functionality separate pieces would have the advantage of making recomposition easier. The data presented here do not contradict that conclusion; perhaps different systems decomposition would reduce the complexities of managing software dependencies.

Software engineering researchers know the problems of working with legacy code. The three organizations described were all developing existing software by enhancing, modifying, and extending it. The systems decomposition had taken place some time ago, and now they were working with a design that had produced successful versions of their products. Research tells us that as systems evolve during development they change their character. It is extremely likely that during that evolution that coupling changes; modules that once exhibited low coupling, may come to have high coupling.

The answer begins with the data; clearly successful software development organizations have dependencies among software components they build. Parnas's expected benefits of modularity have not happened inside the successful development organizations studied. A major contribution of this research has been to assert that systems recomposition ought to become a research topic, and in a sense it already is, because configuration management researchers are currently learning about recomposition issues.

This thesis makes an important step to setting a research agenda for understanding and supporting systems' recomposition. Recomposition has two aspects. First, product needs to be reassembled when changes take place. These changes may come from inside the organization or from external demands made on the company. Second, most product development organizations must assemble multiple variants of the their product for different platforms. This data suggests that currently this takes time and requires developers, teams, and organizations to align their work with each other just to recompose their systems.

8.3 Limitations

This work has some limitations. Currently this data only covers two kinds of software development that are product development and government contracting. Other kinds of software development environments exist as Grudin (1989) has described. Future work ought to examine in-house software development, where the customers and developers inhabit the same organization. Another kind of development context to consider would be commercial contracting. Some of the large product software development organizations use contractors to help them meet release deadlines. These kinds of work arrangements might reveal more complex dependency relationships between the main organization and the contractor.

This study has focused on software during its development. Currently it does not make any connections to the work being done in understanding how to elicit systems requirements and model them. Requirements drive systems evolution. Understanding how these requirements evolve may help to understand how dependencies arise, and may also provide potential sources for better dependency management strategies.

This thesis divided dependencies into three levels: individual, group, and inter-organizational. The individual level and inter-organizational level dependencies seem intuitive. Individual level dependencies involve individuals engaging in articulation work. Inter-organizational level

dependencies focus on the social worlds that software development organizations find themselves in. Group level proves more problematic to define.

The first difficulty arises from the difficulty of the concept of a group. What defines a group? I have tried to separate dependencies that require the participation of the entire group or span functional divisions. However, some of the individual dependencies also span groups. When special builders take on the build management role for a team does that constitute a group-level dependency? The model is a beginning for sorting out the different kinds of dependency management. Further research that explores software dependencies may reveal a better way to organize and categorize the results.

8.4 Conclusions

This thesis has presented a sociological understanding of the practice of software development. It has provided an explanation of how dependencies affect the development of software. Specifically it has shown how technical dependencies among code modules create and reflect social dependencies between the developers, teams, and organizations working on them.

This chapter began with a quote from Victor Basili and John Musa about the need to study the social aspects of software development. It appeared under the heading "software sociology." This work contributes to the broad enterprise of software sociology by offering an explanation of one reason why developers need to coordinate their activities. However, it raises far more questions than it has answered; for example: how can we build technological support for dependency management, what other dependencies exist, and can we design systems to improve dependency management? None of these questions have been explicitly addressed in research yet, but successful software development organizations find themselves resolving these issues temporarily every day of their operation. Software sociology has much to offer both basic research and the development communities.

References

Ackerman, M. (1994). Augmenting the Organizational Memory: A Field Study of Answer Garden. In Malone, T. (eds.), Proceedings of the *Computer Supported Cooperative Work '94*, 243-252. New York, N.Y.: ACM Press.

Anderson, R., and Sharrock, W. (1993). Can Organisations Afford Knowledge? *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 1, no. 3, 143-161.

Babich, W. A. (1986). *Software Configuration Management: Coordination for Team Productivity*. Reading, MA: Addison-Wesley.

Basili, V. R., and Musa, J. D. (1991). The Future Engineering of Software: A Management Perspective. *IEEE Computer*, vol. 24, no. 9, 90-96.

Becker, H. S. (1982). *Art Worlds*. Los Angeles, CA: University of California Press.

Bendifallah, S., and Scacchi, W. (1987). Understanding Software Maintenance Work. *IEEE Transactions on Software Engineering*, vol. 13, no. 3, 311-323.

Bernard, H. R. (1988). *Research Methods in Cultural Anthropology*. Newbury Park, CA: Sage.

Bersoff, E. H. (1984). Elements of Software Configuration Management. *IEEE Transactions on Software Engineering*, vol. 10, no. 1, 79-87.

Bersoff, E. H., Henderson, V. D., and Siegel, S. G. (1980). *Software Configuration Management: An Investment in Product Integrity*. Englewood Cliffs, N.J.: Prentice-Hall.

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, vol. 21, no. 5, 61-72.

Bowers, J. (1994). The Work to Make a Network Work: Studying CSCW in Action. In Furuta, R., and Neuwirth, C. (eds.), Proceedings of the *Computer Supported Cooperative Work '94*, 287-298. New York, N. Y.: ACM Press.

Brooks Jr., F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley.

Brooks Jr., F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, vol. 20, no. 4, 10-19.

Brooks Jr., F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. (20th Anniversary Edition ed.) Reading, MA: Addison-Wesley.

Button, G., and Dourish, P. (1996). Technomethodology: Paradoxes and Possibilities. In Tauber, M. J. (eds.), Proceedings of the *ACM CHI'96 Conference on Human Factors in Computing Systems*, 19-26. ACM Press.

Button, G., and Sharrock, W. (1994). Occassioned Practices in the Work of Software Engineers In Goguen, J., and Jirotka, M. (eds.), *Requirements Engineering* , 217-240. London, U.K.: Academic Press Ltd.

Caballero, C. (1994). Life Cycle: Now the Focus in UNIX CM Market. *Application Development Trends*. August, 1994. 49-54, 64,86.

Compton, S. B., and Conner, G. R. (1994). *Configuration Management for Software*. New York, N. Y.: Van Nostrand Reinhold.

Curtis, B. (1995). Can Speech Acts Walk the Talk? *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 3, no. 1, 61-64.

Curtis, B., Krasner, H., and Iscoe, N. (1986). A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, vol. 31, no. 11, 1268-1287.

Cusumano, M. A., and Selby, R. W. (1995). *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York, N. Y.: The Free Press.

Dart, S. A. (1992). *The Past, Present, and Future of Configuration Management*. Technical Report No. CMU/SEI-92-TR-8. Software Engineering Institute.

Davies, L., and Nielsen, S. (1992). An Ethnographic Study of Configuration Management and Documentation Practices In Kendall, K. E., Lyytinen, K., and De Gross, J. I. (eds.), *The Impact of Computer Supported Technologies on Information Systems Development* , 179-192. Amsterdam: Elsevier Science Publishers B.V.

Department of Defense (1970). Configuration Management Practices, Equipment, Munitions, and Computer Programs. United States Department of Defense, MIL-STD 483. December 31, 1970.

Department of Defense (1985). Defense Systems Software Development. United States Department of Defense, DOD-STD 2167, June 4, 1985.

Dowson, M. (1993). Software Process Themes and Issues. In Proceedings of the *Proceedings of 2nd International Conference on the Software Process*, IEEE Computer Society Press.

Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. (1992). Supporting Indirect Collaborative Design with Integrated Knowledge-Based Design Environments. *Human-Computer Interaction*, vol. 7, no. 3, 281-314.

Fromme, B. (1994). CM Hit List. *Advanced Systems*. November, 1994. 72,74,76-77.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Reading, Massachusetts: Addison-Wesley Publishing Company.

Garlan, D., and Perry, D. (1994). Software Architecture: Practice, Potential, and Pitfalls. In Proceedings of the *16th International Conference on Software Engineering*, 363-364. Washington, D.C.: IEEE Press.

Gerson, E. M., and Star, S. L. (1986). Analyzing Due Process in the Workplace. *ACM Transactions on Office Systems*, vol. 4, no. 3, 257-270.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of Software Engineering*. Englewood Cliffs, N. J.: Prentice-Hall.

Gibbs, W. W. (1994). Software's Chronic Crisis. *Scientific American*, vol. 271, no. 3, 86-95.

Giddens, A. (1989). *Sociology*. Cambridge, UK: Polity Press.

Glaser, B. G., and Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Hawthorne, N. Y.: Aldine de Gruyter.

Goguen, J. A., and Jirotka, M. (1994). *Requirements Engineering: Social and Technical Issues*. London: Academic Press Ltd.

Green-Armytage, J. (1993). Why Taurus was always Ill-Starred. *Computer Weekly*. March, 18, 1993. 10.

Grinter, R. (1995). Using a Configuration Management Tool to Coordinate Software Development. In Proceedings of the *ACM Conference on Organizational Computing Systems*, 168-177. New York, N. Y.: ACM Press.

Grudin, J. (1988). Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. In Proceedings of the *Conference on Computer-Supported Cooperative Work '88*, 85-93. New York, N. Y.: ACM Press.

Grudin, J. (1991). Interactive Systems: Bridging the Gaps Between Developers and Users. *IEEE Computer*, vol. 24, no. 4, 59-69.

Grudin, J. (1994). Groupware and Social Dynamics: Eight Challenges for Developers. *Communications of the ACM*, vol. 37, no. 1, 92-105.

Grudin, J., and Palen, L. (1995). Why Groupware Succeeds: Discretion or Mandate? In Marmolin, H., Sundblad, Y., and Schmidt, K. (eds.), Proceedings of the *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, 263-278. Dordrecht: Kluwer Academic Publishers.

Heath, C., and Luff, P. (1991). Collaborative Activity and Technological Design: Task Coordination in London Underground Control Rooms. In Proceedings of the *European Conference on Computer Supported Cooperative Work*, 65-80.

Hughes, J. A., Randall, D., and Shapiro, D. (1993). From Ethnographic Record to System Design: Some Experiences from the Field. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 1, no. 3, 123-141.

Ingram, P. (1994). The Market for CM Tools. In Proceedings of the *UNICOM Conference*. October, 1994. London. U.K.

Jorgenson, D. L. (1989). *Participant Observation*. Newbury Park, CA: Sage.

Leblang, D. B. (1994). The CM Challenge: Configuration Management that Works In Tichy, W. F. (eds.), *Configuration Management* , 1-37. Chichester, UK: John Wiley & Sons Ltd.

Lofland, J., and Lofland, L. (1984). *Analyzing Social Settings: A Guide to Qualitative Observation and Analysis*. (2nd ed.) Belmont, CA: Wadsworth.

Lubars, M., Potts, C., and Richter, C. (1993). A Review of the State of the Practice in Requirements Modeling. In Fickas, S., and Finkelstein, A. (eds.), Proceedings of the *Requirements Engineering 1993*, 2-14. IEEE Computer Society Press.

Lubkin, D. C. (1991). DSEE: A Software Configuration Management Tool. *The Hewlett-Packard Journal*, vol. 42, no. 3, 77-83.

Mahler, A. (1994). Variants: Keeping Things Together and Telling Them Apart In Tichy, W. F. (eds.), *Configuration Management* , 73-97. Chichester, UK: John Wiley & Sons, Ltd.

Markus, M. L., and Connolly, T. (1990). Why CSCW Applications Fail: Problems in the Adoption of Interdependent Work Tools. In Proceedings of the *Conference on Computer Supported Cooperative Work '90*, ACM Press.

Marshall, C., and Roseman, G. B. (1989). *Designing Qualitative Research*. Newbury Park, CA: Sage.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, vol. 2, no. 4, 308-320.

Nato Science Committee (1969). *Working Conference on Software Engineering*. No. Nato Scientific Affairs Division.

Nix, K. (1994). Using CM. *Software Development*. December, 1994. 61-65.

Orlikowski, W. J. (1992). Learning from Notes: Organizational Issues in Groupware Implementation. In Proceedings of the *Conference on Computer-Supported Cooperative Work '92*, 362-369. New York, N.Y.: ACM Press.

Parnas, D. L. (1972). A Technique for Software Module Specification with Examples. *Communications of the ACM*, vol. 15, no. 5, 330-336.

Parnas, D. L., and Clements, P. C. (1986). A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, vol. 12, no. 2, 251-257.

Perin, C. (1991). Electronic Social Fields in Bureaucracies. *Communications of the ACM*, vol. 34, no. 12, 75-82.

Perry, D. E., Staudenmayer, N. A., and Votta, L. G. (1994). People, Organizations, and Process Improvement. *IEEE Software*, vol. 11, no. 4, 36-45.

Pickering, J. M., and Grinter, R. E. (1995). Software Engineering and CSCW: A Common Research Ground In Taylor, R. N., and Coutaz, J. (eds.), *Software Engineering and Human-Computer Interaction: ICSE'94 Workshop on SE-HCI Joint Research Issues* , 241-250. Heidelberg: Springer-Verlag.

Pickering, J. M., and King, J. L. (1995). Hardwiring Weak Ties: Interorganizational Computer-Mediated Communication, Occupational Communities and Organizational Change. *Organization Science*, vol. 6, no. 4, 479-486.

Potts, C. (1993). Software Engineering Research Revisited. *IEEE Software*, vol. 10, no. 5, 19-28.

Quintas, P. (1993). *Social Dimensions of Systems Engineering: People, Processes, Policies and Software Development*. New York, N. Y.: E. Horwood.

Rogers, Y. (1993). Coordinating Computer-Mediated Work. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 1, no. 4, 295-315.

Rosenberg, N. (1982). *Inside the black box : technology and economics*. Cambridge, UK: Cambridge University Press.

Samaras, T. T., and Czerwinski, F. L. (1971). *Fundamentals of Configuration Management*. New York, N.Y.: John Wiley & Sons.

Scacchi, W. (1984). Managing Software Engineering Projects: A Social Analysis. *IEEE Transactions on Software Engineering*, vol. 10, no. 1, 45-59.

Schach, S. (1990). *Software Engineering*. Homewood, IL: Aksen Associates.

Schmidt, K., and Bannon, L. (1992). Taking CSCW Seriously: Supporting Articulation Work. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 1, no. 1-2, 7-40.

Schwartz, H., and Jacobs, J. (1979). *Qualitative Sociology: A Method to the Madness*. New York, N. Y.: The Free Press.

Sharrock, W., and Anderson, B. (1993). Working Towards Agreement In Button, G. (eds.), *Technology in Working Order* , London, UK: Routledge.

Simone, C., Divitini, M., and Schmidt, K. (1995). A Notation for Malleable and Interoperable Coordination Mechanisms for CSCW Systems. In Comstock, N., and Ellis, C. (eds.), Proceedings of the *Conference on Organizational Computing Systems*, 44-54. New York, N.Y.: ACM Press.

Sommerville, I. (1989). *Software Engineering*. (3rd ed.) Wokingham, UK: Addison-Wesley.

Strauss, A. (1985). Work and the Division of Labor. *The Sociological Quarterly*, vol. 26, no. 1, 1-19.

Strauss, A. (1987). *Qualitative Analysis for Social Scientists*. New York: N.Y.: Cambridge University Press.

Strauss, A. (1988). The Articulation of Project Work: An Organizational Process. *The Sociological Quarterly*, vol. 29, no. 2, 163-178.

Strauss, A., and Corbin, J. (1990). *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Newbury Park, CA: Sage.

Suchman, L. (1987). *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge, UK: Cambridge University Press.

Suchman, L. (1992). Technologies of Accountability: Of Lizards and Aeroplanes In Button, G. (eds.), *Technology in Working Order: Studies of Work, Interaction, and Technology* , London, UK: Routledge.

Suchman, L. (1994). Do Categories Have Politics? The Language/Action Perspective Reconsidered. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 2, no. 3, 177-190.

Suchman, L. A. (1983). Office Procedure as Practical Action: Models of Work and System Design. *ACM Transaction s on Office Information Systems*, vol. 1, no. 4, 320-328.

Tichy, W. (1985). RCS: A system for Version Control. *Software Practice and Experience*, vol. 15, no. 7, 637-654.

Tichy, W. F. (1992). Programming-in-the-large: Past, Present, and Future. In Proceedings of the *14th International Conference on Software Engineering*, 362-367.

van der Hoek, A., Heimbigner, D., and Wolf, A. (1996). A Generic Peer-to-Peer Repository for Distributed Configuration Management. In Rombach, D. (eds.), Proceedings of the *18th International Conference on Software Engineering*, Los Alamitos, CA: IEEE Press.

Whitgift, D. (1991). *Methods and Tools for Software Configuration Management*. Chichester, UK: John Wiley & Sons.

Whittaker, S., and Schwarz, H. (1995). Back to the Future: Pen and Paper Technology Supports Complex Group Coordination. In Proceedings of the *ACM CHI'95 Conference on Human Factors in Computing Systems*, 495-502. New York, N.Y.: ACM Press.

Winograd, T. (1994). Categories, Disciplines, and Social Coordination. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 2, no. 3, 191-197.

Woolgar, S. (1994). Rethinking Requirements Analysis: Some Implications of Recent Research into Producer-Consumer Relationships in IT Development In Goguen, J. A., and Jirotka, M. (eds.), *Requirements Engineering: Social and Technical Issues* , 201-216. London, UK: Academic Press Ltd.