RELATIVE STUDY OF DEPENDENCY MANAGEMENT DURING SOFTWARE DEVELOPMENT

RELATIVE STUDY OF DEPENDENCY MANAGEMENT DURING SOFTWARE DEVELOPMENT

A major thesis submitted to Faculty of Technology
Of
University of Delhi
In partial fulfillment of the requirements
For
The award of Degree
Master of Engineering
In
Computer Technology and Applications

Submitted By:
Meenu Sehrawat
14/CTA/05

Under the Guidance of:
Mrs. Rajni Jindal

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY

DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI

CERTIFICATE

This is to certify that major thesis entitled **"RELATIVE STUDY OF DEPENDENCY MANAGEMENT DURING SOFTWARE DEVELOPMENT"** submitted by **Meenu Sehrawat** towards the partial fulfillment of requirements for the Degree of Masters of Engineering in Computer Technology and Application is a bonafide record of this work carried out under the supervision and guidance of undersigned. It is also certified that the Thesis has not been submitted for any other degree or diploma in any college.

Mrs. Rajni Jindal
Asst. Professor
Department of Computer Engineering
Delhi College of Engineering
Delhi University

ACKNOWLEDGEMENT

DURING THE PERIOD of my study at **Delhi College of Engineering,** I had the privilege to get associated with highly qualified, experienced and learned people who have been a great source inspiration and encouragement to me from whom, I learn a lesson in co-operation.

NECESSITY IS THE MOTHER OF INVENTION, in the society of engineers, a good engineer is defined as the one having ample knowledge and ability to implement the same. We are very much fortunate as this college has converted us into this elite group of people we call as engineers.

I RENDER MY DEEP SENSE OF GRATITUDE to my Project Guide **Mrs. Rajni Jindal**, for her encouragement, help, constructive, critics and guidance. I am highly thankful to her for her keen interest and involvement at each and every step in the completion of this project. I also wish to express great regards towards my head of department **Dr. Daya Gupta** who was constant source of inspiration to me and always help whenever I was in need.

THROUGHOUT MY LIFE I had thankful to her, my parents, my headband, elders, sisters, brothers and colleagues blessing without which I could not have achieved my desire of becoming Master in Engineering.

MEENU SEHRAWAT

ABSTRACT OF THE DISSERTATION

This research provides a new understanding of the dependencies that exists in software system, and how software developers use practice and technologies to manage them, All software system have dependencies because software modules interact with each other, with documentation , with libraries , and with test suites. Software engineers recognize that these dependencies exists, as technical relationship between the components of the system, and have tried to model them as part of there 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 exists between developers, teams of programs, 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 there 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 interpreting 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 understand 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 representation of dependencies, and consequently the times when SCM systems do not provide the required support to help developers cordite there work. This understanding of how a technology support 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

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. 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.

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 calculation 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 Data Consultancy Services, Microsoft, Borland, and Netscape, and others that build hardware as well as systems, including Intel, 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 job lost, and tragically sometimes with the loss of life, In addition to this several other projects failed viz. the Therac-25 machine killed people with high doses of radiation, the baggage handling system at Denver International Airport, these are only some of the failures, those that were expensive or harmed people.

Why does software fail? 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.

This thesis asserts that software development is difficult in part due the relationships that exist between software modules. Software fails to work for many reasons and this thesis describes just one problem that makes developing systems difficult.

1.2 Failure to Manage Dependencies

The Taurus system was sold as a multi-million pound project that would revolutionize the London Stock Exchange (LSE) and decided to modernize its operations by introducing new technologies both within the exchange itself and among the organizations associates with it (Green-Armytage, 1993). The project started with all the optimism of any new venture sold as revolutionizing an industry. After years spent in development the head of the LSE announced that the Taurus system had 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; technically, it was difficult to align the distributed development efforts so that all the systems worked together. Socially, it was hard to maintain communication 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 mange both the technical and social aspects of these dependencies in the production of software systems.

1.3 Research Question

How do software dependencies influence the development of systems?

The research question is divided into three parts:

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

Software engineering researchers have recognized that dependencies exist, but have focused on their technical aspects. Researcher 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. 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-gong management of multiple 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.

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. Modular software development involves breaking down a problem into its logical components and constructing a solution for each part.

David Parnas 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.

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.

This definition of coupling reveals several things.

- Software engineers clearly recognize that dependencies exist.
- They view them in a purely technical way; for example, dependencies exist when variables get passed between two modules, or one module calls calls another
- The difficulties of having dependencies have purely technical impacts; these relationships interfere with making changes, software reuse, or testing.
- 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, 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 take s 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 begin 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.

Chapters 4, 5 and 6, introduce the three types of sites in study: Tool Corporations, Computer Corporation, and Contract Corporations Each chapter describes how that organization (at the level of the individual, group and organization as whole) cope with the dependencies that arise in their configuration management work. Chapter 7 synthesis 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's to change, there is a continual need 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

It begins by introducing the motivation for establishing a discipline of software engineering, and introductions 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 Development Challenges

The phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

Software development began long before the Garmisch conference was convened. Most of the initial systems were non-commercial command and control system; however, companies including IBM had also begun developing operating system software. During the 60’s, companies involved in developing non-commercial and commercial 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 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 organization required increased. The participants argued that these demands for software ware forcing development organizations into situations that were beyond their current understanding and abilities.

The participants also feel 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 feel 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., Somerville, 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 with in 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 feel that current software development remains 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. Requirement analysis definition and elicitation have become important research topics since the conference.

Design and Development

Concerns about design and development permeated the conference. 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, motion schemes for describing the system structures and states, and criteria for design like flexibility, design for change, usability, reliability and completeness.

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 (1998) spiral mode, are a few of the ideas that have contributed and extended this research.

Measurement

The inability of managers to measure process 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 of 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 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 organization 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 form its parts because different cod modules depended on each other and needed to be ordered to reflect that dependency. They feel 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 hare to manage.

In short, the software product is embedded in 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)

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 organizational (and sometimes institutional) boundaries. (Woolgar, 1994)

Whittaker and Schwartz also note the role of dependencies,

There are multiple 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 off 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.

2.5Summary

Software engineering as a domain of research has been active for a short time, since 1976. In that time advances have been made, but many qu3estions 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 this system. (Whitgift, 1991)

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 setting 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 non-commercial hardware systems development. During and after the Second World War the demand s 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 do not keep accurate records of what had been assembled, and rarely do 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 do not have adequate records of part number identification, chronology of changes, nor change accomplishment. (Samaras and Czerwinski, 1971)

Incidents like this made non-commercial 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 do 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 non-commercial environments. During the 1950's and 960'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.

The standard describes procedures for developing software and includes software configuration management procedures. Specifically it defines five aspects of configuration 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, use 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 sot he 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 b 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 also 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 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 does 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 work shop 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 own aspect of the development process such as structured design, and Integrated Project Support Environments (IPSE's), aimed at providing an entire development environment. 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.

Two forces have conspired to make configuration management a viable market. 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 standards for products the ISO 9000 of American owned companies, must buy software from companies certified as ISO 9000 compliant.

In the United States another standard, primarily aimed at the non-commercial 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, applications 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 he 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; of 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 module out, the tool made a new version of the code and prevented others from checking out he 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 wit 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, make files, 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 form changing the same module at the same time, which slowed down developer’s 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 modulate 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 to 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 Explanation 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 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 tractability of this configuration throughout the system life cycle. (Bersoff, Henderson, and Siegel, 1980)

The author describes 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)

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 advice) 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)

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.

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 hey 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 fro 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 modification to the software being built by a programming team. (Babich, 1986)

Configuration management systems are a form of groupware technology, and as well as affecting individuals they require organization commitments to adopt and use. Recently, Nix (1994) drew a parallel between configuration management and groupware, by claiming that tools acted as a communication 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 needed 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. 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, do not accurately reflect their every day practices. Although their informants completed the required documentation it do 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 focused 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 back ground for this work. Researchers who participate primarily in the Human Computer Interaction (HCL) 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). 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). This research demonstrates that this is also true in software development, where technical dependencies create and reflect social relationships among people and organizations.

Ethno methodological 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 Neilson (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 and tm;, 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 mismatches between who does the work and who gets the benefits, 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 working-in-progress and what other developers are doing, as well as providing models of how software development proceeds. Configuration management systems are similar to work flow 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 support 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, and 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 o dependency management it was impossible to generate testable hypotheses. Instead a qualitative research strategy was chose 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. 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, and understand what they understand. (Schwartz and Jacob, 1979)

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 types organizations. 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 selecti9ng 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.

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. This creates a contrast among the three sites, Tool Corporations, product developer, Computer Corporations, primarily builds systems for the open market, but also provides some

customized solutions for special customers, and Contract Corporations these only engages in contract work.

3.7 Methodology: Data Gathering and Analysis Using Grounded Theory

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 hand on opportunity to explore how developers used the tool. These non-participant observation strategies sensitized me to the environment, and helped me to begin to interpret and make sense of the data.

Observation of people in the configuration management group advising developers about practices and technologies provides 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.

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 question 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.

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?

During these descriptions I took note of important parts in the process, of describing dependency management, and I used probing techniques to gather further data about those specific topics.

I used grounded theory, this 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 bei9ng 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 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.

3.8 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)

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.

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)

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 every day 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)

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 do not build systems 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 problems tracking layer coordinates the definition of problems with the actual changes made to the code itself.

Configuration management practices and technologies provide and 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 appear in the data gathered. The articulation of work among developers as they work on individual modules ws 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, s a perspective it do not explain the set of

dependencies that exerted huge influence on software development, so I turned to social worlds.

3.9 Theoretical Perspective: Social Worlds

During the courses 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 with in a single organization, do 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)

In the quote Becker explains how artists depend on both consumers and producers to support their work. This research reveals that in software development world 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 world do not have the same grounding in tradition as those in art world. However, conventions shaped by market forces, the government, and other communities of practice do influence software development worlds and the people building systems. Social world provide and explanation of the dependencies that influence software

development but do not come from within the organization in which the development is taking place.

3.10 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 crates 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 interaction 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 do not 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.

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 knows it's very hard to pick the present from a snapshot.

Tool Corporation, are small development companies, which builds and sells a configuration management tools in the open market. The three levels o 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 Corporation uses to cope with each dependency are discussed.

4.1 Welcome to Tool Corporations!

Tool Corporations are small software development companies engaged in exclusive development

Of configuration management systems since1986. They have established a presence in the configuration management systems market with their own tools.

In the nineties Tool Corporations have grown substantially in size, their average growth figure is varying between 100%-200%. This growth can be attributed to the transformation of Tool Corporations 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 Corporation were rowing marketing, sales, and services operations rapidly, as well as expanding into foreign countries.

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 organization 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 mixes 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 Corporations 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 version of the product. After the release of new products in Tool Corporations reorganize 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 bases configuration management systems to building a PC client to their system marked an important time for Tool Corporations All these had UNIX workstations and several of them had two on their desks. Their last public release of their configuration management system ran o n a variety of UNIX based platforms such as; HP, Sun and DEC. However, Tool Corporation began to shift more seriously towards PC development during my time there.

The first site in my study was a development division of one-CM tool vendor that I call "Tool Corporations," 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.

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 do 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

4.2 Individual Dependencies in Tool Corporations

Life at these Tool Corporations 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 dependences that they experience in their work. In this section I describe 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 checkout copies of the module and makes their changes. 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 created dependencies between them, that take time and energy to resolve. Developers often feel that parallel development represented weaknesses in the product.

Explanations such as these reveal two important issues in the context of parallel development. First, the developers in Tool Corporations 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 cod 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.

Developers use the evolution vies of a module to find out whether anyone else is currently working on the cod e they need to alter. All the developers working on this project can use the evolution vies. This allows them to make decision 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 y the other person. 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.

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 do not go smoothly. While the developers refer to a single activity of merging, they do develop 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. Sometimes when both change the same lines of code, X's change don't include their changes, and their changes don't include X changes, is harder.

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 ad 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.

A lot of time, 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.

The complexity off 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 th 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. So, basically that person will get together with other people and the other person will oversee the merge.

They discuss what they do, 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 merger module. This activity often takes place as a joint merging effort. The developers sit around one terminal and select the lines that should fo 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 ar 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 off 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, in Tool Corporations that means that the nightly build fails.

That's a matter of grouping a couple of different changes together. Like say one changes Module A and Module B, it's a way of saying these two things is related and one can't use this

change without this change. The relationship between pieces of code is a technical change dependency.

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

There are certain times when in order for one to make a fix it spasm several different [software components]. If one make fix in the GUI code but it also requires a corresponding fix in lets say the [languages] interpreter, or the engine, and since it is only one person making the fix one goes ahead and makes the fix in both areas.

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 be carried out in parallel an must also be coordinate, 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.

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.

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 one doesn't mean that he/she is the only person who's going to resolve the problem.

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 corporations provide important contextual information that the developers would otherwise have to gather by asking people. The developers do 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.

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 based upon their knowledge of a section of the product. Expertise dependencies arise in two ways in Tool Corporations First, a developer would find themselves assigned to work on a part of the product that was unfamiliar to them. Second, a developer work on a section of their code that calls a routine in another part of the product tat they don't know. Both situations crop up routinely in software development in Tool Corporations.

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

In Tool Corporations all of the developers, except for thee newest, were considered experts with some aspect of the system. While the older developers may have skills that span several subsystems, newer developers would know about the aspects of the system that they had worked on since arriving in the organization. All developers in Tool Corporation can 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 can do that, and I'm convinced every one of them can do that. A new person has to learn that.

You can also look, if somebody in a particular module is confined within that module is created by the last person who edited it. So, if he confuse about an algorithm and it's crated by [Developer F] then he knows his identity gets it explained. The last person who modified the module may not know bout the algorithm but they can usually point me to the person who does.

In this case the developer had not been with organization for long enough to learn who was an expert in that particulate area. Instead he relies on the tool to reveal the name of the last person to modify the code. He will then ask that developer whether they were the expert or find out from them who were. 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 workloads 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, the managers move people to different sections of the system, as they do not want to have to rely on any developer to get a problem resolved.

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 in Tool Corporations provide this function. Each night this gather 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 corporations 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.

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 ate 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 systems, and then the build manager discovers the technical aspect of the dependency, informs the developers responsible, who then must mange the social aspect of that dependency.

The corporations make 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.

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

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

Sometimes I can tell form 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.

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. One should 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.

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 checkout 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 Organization Memory of Action

Until now, I have only described dependencies that connect developers in the present. However, developers nearly always network existing code, modifying it to fix repairs and add new functionality. 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. Often developers rely on decisions taken and changes made in the past; I call these historical dependencies.

In Tool Corporations the developers manage their historical dependencies and provide an opportunity for developers to work in the past readily. In other organizations, developers refer to previous versions of the source code, but do not use other information. I have already said that the tool relates changes, "problems," to specific changes in the code, through hypertext links. Unlike other systems, such as Lotus Notes, which also has an archiving facility, the organizational memory in Tool Corporation links the changes made with the actual code, the "data" contained in the system.

Design was done through [Lotus] Notes, we do not 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 Corporations experiment really is instead off 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.

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 these organizations 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 solution.

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.

All the time, organizational memory is really useful. If one makes a change in some code it can be hard to remember what someone else did before. You can look at the comments, which tell 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.

As these quotes illustrate, developers at Tool Corporations 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 do not only involve making the technical changes, but also learning about the social context in which that module evolved. Historical dependencies start with relationship between current and previous versions of system components; however, they also create relationships among the developers who worked on those versions.

Developers come and go in these Tool Corporations 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.

There is an activity going on today between [Developer C and Developer D] 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.

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.

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.

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 Corporations put all the documentation and test suites into the system, so technical writers and testers, can also go into the toll and rediscover the missing context that provides a needed explanation.

In this case the context that one sought, not a development context, but a testing one, could not be put together by any single developer. In this case, the tester clearly feels 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 is embedded in the product, appearing on screens and named in commands. The manager assigns a number of developers the task of going through the system and changing all instances of the old mane 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 the entire artifact 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 saves the developers a lot of time. 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.

When development proceeds at a relaxed pace then people usually take the time to explain what they do 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 comment almost meaningless.

When developers do not find the comments useful then they must find other ways to compensate for this missing data. Well, for one thing when one gets task assigned, he knows that task assigned to him, but he still go to [problem reporting system] to find out exactly what this task is about. They only give him a test synopsis, it doesn't really have a description of what the problem is, how he goes about solving it, so he goes 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.

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-system, 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 system at all.

A rarer problem that Tool Corporations 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 do 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 load offers the developers some support in managing the dependencies as I have described above. However, their sense of configuration management, and its importance, as well as the instantiation of that in the tool itself, do cause them to become dependent on the tool and their practices. I will detail two other important ways that the tool and practices of configuration management shape the way that developers understand the development process in Tool Corporations. Developers rely on the tool implicitly. It manages the process, a lot of things, which one normally do on the fly remembering in ones head, it manages. Mainly versions of files, that's the main advantages 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.

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 do not keep versions of the code.

However in Tool Corporations the developers depend on the tool to organize their work using the problem reporting facility that I described earlier. This works well because the developers control the problem reporting system.

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 do 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 decides 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. Developers depended on the problem reporting facility to arrange their own workload, but also observe that the initial display of the problems do not convey all the information necessary.

The developers use the tool as a starting point to learn about the problem. 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 information about what's expected and pointer to other data sources. Sometimes the developer reliance on the too ends up hurting them though.

In this case the developers who have the problems originally assigned to them, and the new developers have 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.

The developer knows how ht layout of software artifacts provided by the tool, as all the developers in Tool Corporations 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 developers must also manage a set of dependencies unique to the interface of the system. Most of the time, especially with the stuff one works 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 thing it's in the interface, when in fact it really turns out to be that other piece of code somewhere else in not working correctly.

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 then the others. They cope with these dependencies by spending considerable by spending considerable time tracking down the sources of the errors. It often leads him into sections of the system that they do not understand, and then they rely 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 interaction 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 Corporations.

Life Cycle Dependencies

Tool Corporations 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 often work on different platform releases ate the same time.

The 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 for all of Tool Corporation's code.

At the time the developer decides to isolate their 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 do not necessarily want to even conduct a nightly build, but work on big section 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 the at their own pace before merging back into the faster pace main project development. As one of the developers explained,

In both of these cases the tool helped the developers to reduce their dependencies on the pace of development. In worked both ways too. The developers working on testing and fixing the main development do 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 on the project and the new platform do 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 do not provide any mechanisms to help the developers visualize the system as a whole. The fact that this "big picture" are missing and that the developers could not easily put it together.

When development starts on the configuration management tool, Tool Corporations assigns a few developers to the task. At this time the development process is managed by a group of friends, who talk over the walls of their cubicles and in so doing stayed in touch with all the aspects of development.

As the product becomes successful, the company grows and the development group also got larger. The developers begun to specialize in their own sub-systems and lost their sense of the whole system at the same time. While the developers understand their own sub-system well, and many had worked on different sub-system, the developers do nt know the conceptual structure of the product being developed. This problem is exacerbated by the changes in design that occurred throughout its development that alters the system. Developers, who have not been with the company for long, feel the isolation, and that affects them.

There are a lot of different sections, which call routines in other sections, that's sort of, one very much need to know the big picture of the group, as well as the individual part. They may have routines which make calls, type creation calls go to some of my routines, hands me information they need. So one very much has to know what other people are doing, although its kind of scary because everybody works on his own particular areas and no one really knew what it was going to look like when it all came together. So, the should have a lot of design meetings which is really good, that is really important in the beginning to et a good picture, but once one gets specifications, they start into developing our little portions.

Their concern is that no one really understands how the system will look when it will be all put together. Although the system pulls together code for the nightly builds the developers couldn't visualize that whole. Because they could not see that whole, they had trouble recognizing how the parts would fit together, what those interactions would be. It also cripples their ability to practice software techniques such as re-use. Because the developers do 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 do not really understand how the system would fit together.

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 one can 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.

The problem reporting facility though tracks the changes in the system at a very low level, the modifications to the code. The developers want a more generalized description of what their colleagues are doing.

Few developers draw 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 hat the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems. (Garlan and Perry, 1994).

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.

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 do use e-mail to initiate discussions about the system, but these discussions competed with all the other e-mail to initiate discussions about the system, but these discussions competed with the other entire e-mail message, and were often ignored by other developers.

Other developers then to the documentation, most people don't have a clue what anyone else has doe, even at the highest levels. However, during initial phases of his development the documentation often do not exist, or remains out of synchronization with the development effort. Also the focus of the documentation was o users, so developers got little information about the implementation of the features described in the documentation.

Once one has about ten people in a development organization he rapidly realizes that he don't knows what every body else is doing. One can't know what everybody else is doing; when people come from big companies they are very comfortable with that.

As a group the developers at Tool Corporations 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 do 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 wit the development group. In Tool Corporations this relationship grows steadily closer because the developers go form 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.

However, I have elected to treat the testing groups as essentially separate groups from the developers. They perform a different function from the developers. They are primarily responsible for comprehensive system testing, particularly integration test, 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 evolve throughout development, so they must evolve their own checking programs.

The reason whey 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.

The situation gets more complicated because Tool Corporations have a variety of stage 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, they system also becomes used internally.

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, te test scaffold, inside the configuration management tool. This helps them to keep track of which test scaffold belongs o which version of the tool.

At the same time they also need to know 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" – system that have a high degree of compatibility with other systems available in the marketplace – this has become even less true. Tool Corporations, like all software development organizations, needs to maintain string connections with other organizations on their software development world. In this section I describe two external dependencies that influence Tool Corporations, relation ships with vendor's customers.

Vendor Dependencies

Tool Corporations 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 b a large software development organization. The management in Tool Corporations realizes 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 system, so that they can test their product out on all these different combinations. Tool Corporations 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 corporations 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. In Tool Corporations other vendors played an important role in determining when Tool Corporation began developing another version of their system.

As well as influencing when Tool Corporations begins development on a version, other vendors shape what exactly is development. If substrate technologies, like hardware platforms, ad operating systems, do not contain certain functions, then Tool Corporations 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 Corporation, depend on these substrate technologies to offer fast and reliable ways of passing information between databases. Some senior management feels that current commercially available technologies do not do a very good job of this.

Customer Dependencies

Tool Corporations and the potential and existing customers depend on each other. Tool Corporations depends on its customers somewhat to define the requirements for the product that they build. The customer depends on Tool Corporations 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” system, like Tool Corporation’s product, need t support a variety of integrations between the core product and other system. 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 integration, that involves basically taking their tool and taking whatever other product tool that their customers want to use and integrating it with the product.

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 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 one looks at the problem, one tries to duplicate

it, unless its obvious and then reviews it. From in review it goes to either assign or deferred, if it's a problem that one decides can not fix or decides that it doesn't make that much difference. Otherwise assigns it to the right developer.

Tool Corporations 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 Corporations 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

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 Corporations recognized this. In those Tool Corporations where 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 bets-versions for testing, they recognize that customers depends on what they do.

Tool Corporations and its customers and potential customers depend on each other, for requirements, future directions, and support. The organization handles these dependencies n 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 need 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 of the existing system. At the same time the customers depend on Tool Corporations 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 in Tool Corporations.

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 do, and I believe that this fake separation contributes to the fact that dependencies remain ill understood in academic circles.

Dependency management forms a critical part of software development work in Tool Corporations because the tool supports some of this vital coordination work, I did not discover

the importance of managing these dependencies until I studied Computer Corporations that I discuss in the next chapter.

Chapter 5

Case 2:

Large Computer Manufacturer Seeks Good CM System

What you really have is 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 weird, really hard to even visualize what it might look like or even.

Tool Corporations 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 in Computer Corporations and extends the analysis of dependencies by providing supporting evidence for their existence and the forms that they take.

Also Computer Corporations 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 Corporations confirms this assertion and shows that the complexity of dependency management rises as well.

5.1 Welcome to Computer Corporations!

Computer Corporations are large computer companies these initially 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.

This company has enjoyed a period of financial growth in nineties, with profits steadily increasing and sells its products and services to a variety of industries including telecommunication companies, manufacturers, health care ad insurance industries, and banks. The products and services that the sell are high performance real-time data processing. Software development has a different character in Computer Corporations than it does in Tool Corporations.

Recently most of Computer Corporations have expanded their operations to take account of the changing nature of the market place in which it competes. Initially, these companies 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 commuter systems of theirs in place. As a reaction to this phenomenon Computer Corporations have moved towards open systems and now provides integrations between its won hardware and other commercially available systems.

Like Tool Corporations, Computer Corporations sells its products on the open market, however unlike Tool Corporations; they also offer customized versions for special customers who can afford them.

The impact o open system 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 works for the local environment. The system is relatively sophisticated tracking the development of different versions of the product including customer-specific changes. However, priorities change within these companies and instead of maintaining the internally built system the companies decide to buy a commercial tool. This has the advantage of then freeing these 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.

5.2 Individual Level Dependencies

Parallel Development Dependencies

The developers in Computer Corporations like those in Tool Corporation engage in parallel development. While the developers in Computer Corporations might have been experiencing adoption problems with this particular merge tool, those in Tool Corporations were familiar with the technology. However, because the developers in Computer Corporations were new to the merge facility they revealed important aspects of parallel development dependencies.

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

The scale of operations in Computer Corporations means the developers often find themselves in parallel development situation either with several other developers all working on the working on the same file or even or working on multiple sets of changes on the same file themselves.

If there are more than two versions out there that are in parallel one has to make sure that what one actually merging is right one. And if they have a three-way merge then one has 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 one makes comparison between the two, diff. between the two files. And then if one knows that the line of code very well one will say, oh these are the changes that appeared, especially if it is other person who knows correctly labeled nobody else shouldn't say. It's all the process which one has to make sure.

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

Code freeze refers to a point in the life cycle when no more changes can be made to the software. This proceeds 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 ay developers reach code freeze and testing, they become much more concerned about the changes that they make to software. The test and fix cycle creates a grate 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 crates eve more stress, because it requires that developers realign their efforts with each other, and cuts into very tight schedule.

The developers at Computer Corporations have a number of strategies for copying with their parallel development situations. The developers ate Tool Corporations often discuss the poor division of labor that they believed created the parallel development problems. In Computer work up so that the product has conceptually distinct areas.

Unlike the developers at Tool Corporations the developers at Computer Corporations realized the limits of a good architecture. Software what gets divided into conceptually distinct sub-system may still have a sub-system that end 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-system, which means that people working on kernel software must maintain more relationships with other groups than any of the other groups. It crates an added complexity for kernel software development.

A good software system may have a central kernel that orchestrates the functioning of the entire product. Dividing the labor into the different sub-system works well for the people who work on peripheral sub-system 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 ding out whether the work of others ma impact their own.

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 in Computer Corporations 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 Corporation’s product, but the developers also use 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. These who used RCS had to perform manual merges using editors like VI and emacs, and the UNIX “diff” facility as RCS do not provide a merging facility. However, like the developers at Tool Corporations, and perhaps heightened when using RCS doing manual merges, the developers viewed merging as a difficult and time consuming situation to be avoided.

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. Developers have hopes that the new tool

would reaffirm their own priorities of how good development takes place, by code reviews, adequate testing and planning for maintainability. However Tool Corporation's product do not give any priority to merging and in the absence of that, typically the merge gets left to the last developers to finish doing.

Another problem that the developers have with the new tool was that it do not appear to accommodate their current working practices. The fact that new tools require adaptation, and that they involve changing work practice, is not a new discovery.

When the developers discovered that the tools do not let them save a semi-merged module hey got quite concerned. Alpha, the in-house product that the 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 module, the developers merged large sections of the product together. Under this paradigm of thinning 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 Corporations product required a different merging strategy based on a lower level of granularity. The teacher's paradigm was "merge little and merge often."

In Computer Corporations 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 crated 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 sb-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 Corporations must also manage change dependencies. Despite the differences between the methods of assigning changes to developers – Computer Corporations do not use the same problem reporting facility as Tool Corporations, 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 Corporations implies that technical change dependencies may diffuse across the product and as a consequence across the organization. Infrastructure software ha lots of dependencies n lots of other bits and pieces of infrastructure um and bits and pieces of operating system because of that there are lots of issues on trying to keep data structures in synchs trying to keep changes in synchronization um as well as the general issue of supporting multiple threads. The customer driven requirements o being able to support multiple releases with multiple threads goes on in parallel.

In tool Corporations the developers usually talk about these changes in both technical and social ways, describing changes in other sections of the code and who was working on them. In Computer Corporations the developers rely 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 Corporations at the point when the product was assembled.

The developers in Computer Corporations face additional problems to trying to make changes to the code due to the size of the development effort. Even though the developer wanted to make a one-line change they need to ensure that it works with all the code that interacted with the revised module. In Tool Corporations the tool provides the developers with a stable base from which they could all begin working on the related changes. Developers at Computer Corporations do 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 a finding a stable baseline from which to start making changes, developers must synchronize their changes.

The developers in Tool Corporations rely heavily on the tool to help them synchronize their changes. As a small group using the too 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. The situation is vastly different in Computer Corporations when the developer works on changes that span different sub-systems; they find themselves working on modifications those 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.

Changes dependencies, like parallel development dependencies, were also in the midst of transforming with the adoption of Tool Corporation’s product. Alpha used a project – a collection of modules—as its unit of operation. Tool Corporation’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 Corporations rely 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.

Developers 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 our sometimes.

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 tat they end up working together.

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 he work of one will certainly impact the work of the other.

Integration Dependencies

In Computer Corporations 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 managers in Tool Corporations the build managers in Computer Corporations use manual strategies for gathering latest changes

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 one can really find it is in the build instructions of the make files one can see some of it but it's not clear there what all the dependencies are.

Although one can track some of the technical aspects of integration dependencies, using build instructions and make files. She still relies on the manual specification of these relationships. When developers get that wrong, then they have to take the time to find out where the problem is and fix it. This involves communication with the developers and finding out what they know, and then building the whole from the parts that each developer understands. Actually a build manager has a big role to play, he is one that knows mostly the structure of the system especially. The build manager can see but still has to ask developers. 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.

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

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

One of the reasons why integrating 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.

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 Corporations rely on the tool.

The builders in Computer Corporations help individual groups to integrate their sub-systems. At the same time, along with the release area group the builders provide a broader function.

Literally it takes, there's if one looks at the people involved it takes a lot of people just focused don their work problem. No one person can actually release anything it takes around 30 people in one group to the release area. While others go around trying to figure out what the configuration is, and how to build it, and why this build failed, that's all they do.

As well as having large teams of people, the organization needs to routinely integrate the entire product together. Building begin this process by gathering individual sub-systems together and trying to figure out what other parts of the product their sub-system relate 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. 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 are incompatibilities because everything was not really integrated at the same time.

These are the problems in building the sub-systems that leads one 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 make file resolves.

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

Development do 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 sometimes from last week because it's not here in release. It's not there. So if we could get a version scheme into release then one can 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 one can only release during those times.

Computer Corporation 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 do 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 between Computer Corporations and Tool Corporations to resolve integration dependencies between developers Computer Corporations

provide special people within the organization, builders Computer Corporations also have an organizational division, the release area group, to work on a level of integration that Tool Corporations do not deal with at all. What one developer can do in his spare time in Tool Corporations require an entire department in Computer Corporations? As the scale of development increases, the complexity of dependencies rises.

Historical Dependencies

Like their counterparts in Tool Corporations the developers at Computer Corporations also rely on decisions and changes made in the past; however, unlike Tool Corporations Computer Corporations does no t 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.

The developers use a combination of tools to help him see the differences between the two versions of code that they are interested in because they do not have access to a merge facility.

They then have to work out what those differences in the code implied, what was the change that caused those particular differences? Sometimes however, this proves too time consuming and the preferred mode of working was to hope that his general knowledge of the software would ensure that they do not break anything while making his changes.

The developers who have switched over to Tool Corporation'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.

It was only a simple request, that the tool display the nearest ancestor rather than the furthest ancestor when involved, but when the software evolves through several hundred versions, still showing this distant relative will be very frustrating.

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 developers want another way of visualizing the evolution of the module, perhaps following various changes as they evolve. The developers in Tool Corporations never mention reorganizing history, and I speculate that they do adopt 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 in Computer Corporations use to manage the historical dependencies they have with others.

Configuration Management Tool and Practice Dependencies

The developers at Computer Corporations use a variety of systems to support their configuration management activities. I have already discussed how these come to rely on those tools. When they are changed then 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 at Computer Corporations generally value 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 the make errors.

Being able to go back in time is nice because providing developers know exactly what they can go back to, they can say add a path as existed at this point in time, so they have all old modules even leading in the past, the future from this point, even in the (operating system) they're there so they don't have to, they will never have to have any of these worries.

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.

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.

In Computer Corporations those individuals responsible for helping the company to move to the new product are very busy. Often they do not have time to spend helping individual developers,

Or groups work out policies for tool usage. This leaves the developers wondering how to incorporate the tool into their work routines.

Platform Dependencies

One dependency that the developers at Computer Corporations encounter that do not affect people in Tool Corporations involves different hardware platforms. In Computer Corporation they develop their product for one particular hardware platform that Tool Corporation's tools do 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 created 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.

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 ru 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 Corporations build software to run on some platforms that Tool Corporation's tool do not run , because they offer certain feature that are important in Computer Corporation'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 heir 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 in Tool Corporations different platforms have different release cycles. In Computer Corporations 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 product are actually more in the QA life cause that's when you integrate everything and you test the integration of all these things.

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 tow releases. So at any time, Computer Corporations have 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 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 in Computer Corporations. 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 Corporations builds.

However, the developers in Computer Corporations still need to understand the "big picture" of their own sub-system. 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 will do.

The scale of their sub-system easily matches the size of the all products in Tool Corporations, and as such these problems were comparable. However, I notice a marked difference between the expectations of developers in Computer Corporations and those in Tool Corporations Developers in Tool Corporations feel 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 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 grows bigger more people work on different parts of the software and the developers slowly lose track of what everyone else was doing.

In Computer Corporations while developers wanted to understand the big picture, they do not view the fact that couldn't as a managerial problem.

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

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

In Computer Corporations 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 section into sub-system that different group 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 product, in technical terms, and then assigning the work to appropriate groups.

An architects work consists of the translation from the direction for the product, into logical changes required, into technical changes that need o be made. The architects recognize that they also manage these dependencies that they define and crate for the groups. They become boundary spanners, helping groups communicate with each other.

At the same time that architects define, create, and sustain technical dependencies between the different groups involved in building the products in Computer Corporations they also engage in social dependency management. They move between groups presenting each with the bigger picture, an understanding of what the otter group does, and why that's important. Garlan and Perry describe these social properties in dry term, but for practicing architects, conveying these meanings, building these shared understanding, 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 th process of modularity the emphasis focuses on

“separation of concerns” dividing up conceptually distance parts of the system. For example, it would be bad practice to confuse kernel function 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-system 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.

Some developers in Computer Corporation feel that he problem was particularly bad within this particular product because it was designed to have many shared-code dependencies. 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 hat the sub-systems operate together developers from different groups need to have access o 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 pt their code somewhere where they can find it. This turns out to be more problematic than it sounds. Developers have a number of ways of trying to find and maintain their knowledge of where that code resides

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 contact across h 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 end s up being in the situation of not knowing who’s using their code at all.

Their strategy for managing their end of the dependency consists of sending out e-mail letting people knows when radical changes will occur. Often the e-mail needs to be broadcast to the entire company to be sure hat everyone who uses the code will be aware of the changes that are about to be made to the tool. A manager explained hat for some of the more common pieces of shared code get managed by a central group.

Supporting these dependencies by hand is error prone, and so Computer Corporations have an initiative to support these dependencies using technologies, that include Tool Corporation’s product. However, these kinds of dependencies cannot be supported easily with technology.

Tool Corporation’s product relies on a database to store all the system components. The developers in Tool Corporations use their tools, and its database to store their code. In fact they use 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.

In Computer Corporations the scale of the problem is significantly larger.

The issue they have with it that inside the database it’s very good at that. Their problem is beyond the scope of a single database. They are 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.

Computer Corporations also runs 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 no 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 abut individuals anymore you know the corporations doesn't see X who's working on this product, they see the product, right, and they want to deal with it at the product level not at X's level. So when they see something show up in the central corporate repository they don't want to see stuff owned by X they want to see it be owned it be owned by the product which is the group right.

Having individual names associated with sections of code may provide some information for other; 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. Developers where they talk 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 the create the desired functions of the product; sub-system interact with each other to provide the desired functionalities and outputs. At the same time they crate dependencies between the different groups working on building those relationships between the codes.

5.4 Inter-organizational Dependencies

Like Tool Corporations, and Contract Corporations, Computer Corporations do not develop software in a c 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 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 Corporations must manage as a commercial software vendor in the high-performance real-time systems market.

Vendor Dependencies

Computer Corporations depends on vendors in two ways. They depend on the vendor of the tools that they use in their development environment. Computer Corporations also depends on other vendors with whom they compete for their business.

Computer Corporations chose to bring Tool Corporation's tool into their development environment to help them address their configurations 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 companies to review the ways that they had released in source code form. This led the companies to review the ways that they currently manage their development process, and the role of tools such as RCS and particularly the homegrown tool, Alpha.

They opted for bringing Tool Corporation's product into their development environment as the saw advantages in doing that. As 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 and by bringing in a vendor, Computer Corporations could benefit from upgrades.

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

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

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

These vendors change their code, not only in terms of functionality of individual module, 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 developers spend much of their time reworking heir code to accommodate that new change to the integration. Although they may hear rumors about potential changes coming, they must often adapt his entire development schedule simply because a vendor produces a new upgrade, or worse yet, an entirely new release. They depend on the actions o other developers in other organizations routinely, because his code depends on the software developed by the other company.

Customer Dependencies

Computer Corporations manage very direct and more indirect customer dependencies like Tool Corporations Computer Corporations 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 Corporation's product will expect support for a certain time after they buy the software. Organization that contract with Computer Corporations may demand that they guarantee support for a certain length of time after the customized system is delivered.

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

Computer Corporations ay wish to innovate by design and develop completely new product. However, at the same time do customers depend on Computer Corporations. 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 Corporations take in their own development. These influences are less direct, and to my knowledge Computer Corporations do not have direction meetings with preferred customers like Tool Corporations Customers exert these influences in two ways; what they want, and when they want it.

The influences that customers have on determining what features should be in Computer Corporations product and the implications that these demands have on the development of the product over time.

In this case the two groups (X and Y) had diverged over the ears 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 is 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 of new releases. However, this is not strictly a customer dependency, because other vendors releasing rival products also force Computer Corporations 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 Corporations and its rivals produce. This demand has begun to put a huge stress on Computer Corporations.

Customers may directly depend on Computer Corporations for enhancements, features and support. At the same time Computer Corporations must follow the demand s 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 Corporations about what to provide, if Computer Corporation do not supply a certain set of those features then the customer may choose to purchase new systems from a rival.

5.5 Summary

Research in Computer Corporations reveals that the organizations share many of the same dependencies as Tool Corporations. They must manage parallel development, integration, expertise, and change dependencies. However, the scale of operations, the size of the product, the number of developers strategies for coping with these dependencies that are embodied in the organization itself. Unlike Tool Corporations, Computer Corporations need 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 Corporations have distributed development environment unlike Tool Corporations. The main development operation consists of several large buildings and they also have developers working I 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 information communications networks work well enough so that the company can release software. However, Computer Corporations recognize that this method are susceptible to failure, and has begun to pursue other course of action.

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

Chapter 6

Case 3:
Non-commercial Contractor Adapts to Policies

What type of tools do you nee, 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 any non-commercial 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 jus make a minor improvement that they're going to throw away in ten years.

This chapter covers Contract Corporations small contracting companies. I review interorganizational organizational and individual dependencies that impact the software development process. The emphasis is on highlighting both the similarities and differences between a non-commercial environment and a commercial one.

6.1 Contract Corporations

Contract Corporations compete for both non-commercial contracts to supply customer specific hardware and software systems. They are specialized in providing application specifics system for tracking control centers, and large data handling systems.

Despite being a publicly held company, it remains difficult to find out exact details about this company. The division of the companies under study concerns itself with non-commercial contracts exclusively, ranging from unclassified projects.

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. This coupled with the sensitive nature of working on government contracts meant that the data from this site is limited. The kind of dependencies that I was able to find with most of them concerned the social worked of non-commercial 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, non-commercial contracting environments provide an important contrast class for the previous two chapters. Large scale software development also begun in the non-commercial and government context, and many configuration management text books acknowledge this heritage some focus exclusively on non-commercial software development standards so I believe that the opportunity to revisit configuration management in this context proves worthwhile.

6.2 Inter-organizational Dependencies in Non-Commercial Software Development

This section describes the inter-organizational dependencies Contract Corporations manage. These dependencies differ from the kinds that Tool Corporations and Contract Corporations manage because of the non-commercial 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 vendor. In the past, non-commercial systems may have been free of commercial products but both today. Commercial Off-The-Shelf (COTS) development comprises a large part of non-commercial development. Sometimes non-commercial applications center on a COTS product, and then the in-house development involves building security, process, and other non-commercial specific wrappers around the product. Other times contractors wish to incorporate certain functionality into their system, and simply build integration to a COTS product. Finally, contractors may choose to use COTS products in their own development environment to help with the project. Often non-commercial 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.

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 off the product, they depend on the vendor to provide the appropriate behavior. However, non-commercial systems have certain safety requirements that make relying on commercial products very dangerous, financially, and in terms of lives.

If you go fro one version of the compiler to the next usually there's new switches, typical thing is optimizing, your code 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 changes, CM systems have changed, binary files have change, binaries have changes, 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?

Farsighted developers and project manager 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 is explained:

Far searching compiler business will knew about future improvements so the compiler can automatically take care of things, or take advantage off 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 it can be detected as to when that processor is there and do it different. So just having different hardware can sometimes affect your binaries coming out the end.

Commercial vendors have their own priorities and demands. Because Contract Corporations have chosen to use COTS products, they bow 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 configurations management, as well as knowing about all the hidden affects of changing one component in their development environment.

Whether these use the COTS product in their development environment, or bundle it into the system that they provide to the customer, contract Corporations has to cope with having been sold 'vapor ware." In this case they solve it but at tremendous personal cost to themselves.

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 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 often they do that, if it's aggressive company, nine months, every nine months they ship a new modifications, over ten years that's 12 that's 13 retrain and that all has to be coasted as expense.

Contractors find themselves both at the receiving end and in the middle of dependency relationships with vendors. They rely on vendors' products I 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 Corporations faces are those where a change in one aspect of the environment involves a hidden change in something else. Sometimes Contract Corporations do not even go to this point, because the products that they bought do not do what they thought it will do. Then they must either buy another product or fix the final customer. At the same time that Contract Corporations 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 non-commercial customer has in software development.

Standards

Standards influence the way software gets developed in very obvious ways. They dictate the exact format of the solutions. 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.

Unless it is a large organization like the government that mandates you shall do this, you shall do that, and these soft sciences are almost totally unknown.

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 fluctuated from contract to contract. Sometimes standards create extra layers of work for Contract Corporations 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 Corporations. In both cases Contract Corporations do not make that

decision. Although they may choose to follow certain practices out off 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 Corporations developers may have problems implementing them.

Requirements

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

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

In Contract Corporations 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 Corporation's hiring strategy for that particular project. As a result of this, Contract Corporations 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 Corporation's approach to development. Money shapes the development in a variety of ways though, some of which I have already discussed in the contest of other dependencies. Companies like Contract Corporations usually bid for contracts that they want. In their bid, Contract Corporations need to demonstrate that they can provide the solution for a price that the customer can actually afford. Although this sound easy, in practice the customer do not often provide information about exactly how much money they have to spend on the contract.

When Contract Corporations puts contract bid together, money do not solely determine what they propose to provide in the solution.

At the highest level one has what is called the end safety and at eh lowest level one has rupees. There's a spectrum in between of all kinds of software typically commercial systems are closer to the rupee impact, if the software fails one finds a solutions, what's the impact, generally something that affects the bottom line. Contractors like Contract Corporations know that when a customer need s a safety critical system they will pay to have certain safety enhancing features in the final solution.

Multiple Contractors / Contracts Dependencies

Non-commercial command and control systems often contain millions of lines of code, usually embedded into non-commercial specific hardware platforms. Usually these contracts require that multiple organizations work together to produce the final system, and Contract Corporations find itself often working with a variety of other companies, large and small.

In 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 configurations 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.

In this situation the development done by Contract Corporations are dependent on the demands of the governmental/non-commercial agency. However, the central coordinator was not stable during the project. Large systems developments often take several years to complete and during that time the people at the contracting organizations and the government change.

This contract exemplifies the problems of working in a non-commercial 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 contractors 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.

When Contract Corporations get 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. Non-commercial contracting environments find themselves in the position of being heterogeneous because previous development contracts have produced very diverse environments.

Contracting environment 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 Corporations cope by hiring and assigning engineers who can transition between these different languages and platforms

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 become responsible for ensuring that the division of work minimizes 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 context, they have to start with existing systems and extend or modify existing code and these existing stems have been built by several contractors.

These studies on Contract Corporation's strategies for coping with these impacts. They relied people with tow critical skill sets. Their preferred developers had working knowledge of a variety of languages and hardware. Second, Contract Corporations also wanted people who could move around between projects, software and hardware easily. Although the project managers recognize the costs of moving people abruptly between these different environments they sometimes have no choice.

The challenges of heterogeneous environments surfaced in Contract Corporation's hiring strategies. The company often sought people from small organizations, using the rationale that people in those companies would have to do man 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 non-commercial refers to the environment as black or classified. Classified environments create special demands on contractors like Contract Corporations The special security measures that the non-commercial demand to keep the project secret impact Contract Corporation'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. These also maintain a careful log of visitors and visitors' badges displayed the level of security clearance that the person visiting has.

Classified environments crate special demands on contractor' 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 operations.

What happens if one is in the classified environment where the system doesn't change, one is prohibited form changing it. Everything has to be exactly the same today as it will be twenty years from today. The systems do 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 the years, twenty years from now, they call, generally those are black environments running. Classified environments in the 1990's ay then still use technologies form 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 supports 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, he 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.

If it's a government operation so that they just ire additional government people that'll work there for the next 20 years, and part of out 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.

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 environment 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. To recognize that both action and inactivity shapes 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 system 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 develop with are actually the same tools that they'll maintain with

I described the extremes of classified environments previously, but all non-commercial environments have much longer lifetimes than vendors like Tool Corporations and Computer Corporations deal with. Even in non-classified environments, contractors like Contract Corporations 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 o people who are going to be using the system 10 years form 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 f 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.

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 non-commercial contracting environments must cope with to develop software. In this section I describe those dependencies and their affects on the software development process.

Historical Dependencies

In both Tool Corporations and Computer Corporations developers routinely go back to previous versions of the code to find out what changes other developers made. In Contract

Corporations they have another reason to visit the past, to recreate environments that they must extend and modify. Like all software non-commercial systems grow old and become obsolete. Contract Corporations 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 non-commercial software can grow very old before it gets replaced. Whereas computer corporations and tool corporations developers would go back routinely a few months or usually three years contract corporations sometimes found themselves regenerating environments from twenty years ago. This following quote illustrates the problems of working in so far the past. In case contract corporations need to regenerate a binary, X and when they compile the code they got X instead. At first they assume that X had been developed by different source code.

In this case the developers had gone to the lengths of recreating the exact development environment to recompile the software to reduce changes that have 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 do not require intensive investigation as to how X was developed.

This is an extreme example of the difficulties of working with what has been called legacy code old pieces of the system. In tool corporations developers depend on the work that they colleagues did in the past. Contract corporations take this to the extreme because of the circumstances of non-commercial contracting situations Contract Corporations often finds itself working with the 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 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

In contract corporations the developers depend on their configuration manager the embodiment of their tools and practices. The configuration manager takes responsibility

For ensuring that the developers and managers produced all the required documentation

for the project. They depend on both groups to get her own work done, and spent considerable time adjusting their working practices, communicating with, these two groups. In this section I describe how their technical work of producing documents generated dependency relationships.

Non-commercial contracting environments following standards like DOD-STD 2167A must produce many reports accompanying the software and hardware that they develop. 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.

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 a job, the configuration manager needs the developers working on specific sections of the project to complete various documents. To know what is needed from developers strategies are devised to find out. Particularly surf the network for documents to see where the developers were in the documentation process. So that happened, whenever one shipped a tape they will keep the electronic copy and maintain that version.

These strategies helped configuration managers to align their own efforts with those of the developers. As CM relies on them completing detailed descriptions of the software that they had developed to complete own required work, also needs 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, they also rely on the project leaders in there document production work. Project leaders for the contract change a number of times in Contract Corporations while they remained as configuration manager. Each new project leader brings their own preferred ways of working, with respect to document production.

These changes in systems impact the work of the configuration managers because they have to learn new products and devise new ways of producing the required documentation. The technical dependency between them and the project leader involves the production and verification of the 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 in Contract Corporations face the hazards of changing code once they've built the system. Hiring and financial incentives these use to encourage developers to remember as much of the code as possible.

Clearly developers at Contract Corporations 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? Study from Tool Corporations and Computer Corporations suggest that this may be a problem for Contract Corporations. It may also not only affect Contract Corporations but other contractors working on the same overall system, but different, yet dependent, sub-systems.

6.4 Summary

Software development in the non-commercial contracting context highlights the complex set of dependencies among contractors, customers, and vendors. Building software systems involves managing these relationships. This case demonstrated 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 in Contract Corporations also have unique features. Standards used in non-commercial contracting focus greater attention on documentation activities. Much of the work that the configuration manager describes 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 complex dependencies so it's a layered dependency problem so its 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 its just a dependency tree that's really um weird, really hard to even visualize what it might look like or even 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 enterprises 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. Its solving that dependency issue. Configuration manager Computer Corporations

The previous three chapters presented data gathered from different software developments 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 the 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. Software engineers focus on the technical aspects of that problem, for example: does he notation produce unwieldy specifications for complex systems, do this testing strategy work in very large cases, and do 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 Corporations can achieve with 14 developers, one part-time build manager, and a project manager, takes Computer Corporations developers, people employed full-time as builders, steering committees, and entire departments to accomplish.

The nature of the solutions that Tool Corporations and Computer Corporations have adopted to manage these problems differs as well. Both organizations use informal procedures, e-mail, and face-to-face communications 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 Corporations 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 Corporation s and Computer Corporations concerned the homogeneity of tool usage. In Tool Corporations 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 Corporation'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 in Computer Corporations 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 in Computer Corporations although the environment consisted of seemingly few platforms these different platforms supported a number of different configuration management tools. n 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 do not bridge the entire development effort, you to remain localized around certain features of the tool that they used.

The number of repositories for code in Computer Corporations is unclear, but n their vision of the controlled development environment the configuration management group

Estimated that 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 the 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 the tool usage. Critical mass theories suggest that adoption of a tool requires that a certain fraction of participants use the technology. In tool corporations everyone uses the same tool. As everyone uses the same tool the value of the kinds of the coordination that they could achieve increased. Beyond the obvious emphasis that tool corporation developers placed on configuration management two factors made this level of usage possible and the scale of the system itself and the penetration of the tool into the environment.

In 1995 Grudin and Palen 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 at Microsoft and suns at suns. This study suggests that whether an organization has one or more platforms do not matter if the tool works with all of them. Developers working on different platforms in tool corporations managed to work with each other because the tool worked on both platforms. However the developers in computer corporations have 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. In Computer Corporations the developers had begun to adapt their existing practices ass they switched over to the new tool.

However, in Tool Corporations t hey 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” solution, 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 inter face design: in-house, product and contract. In this study I have examined two different contexts: product and contract. This study suggests tat the context 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 Corporations, like Tool Corporations and Computer Corporations, 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 base don the context of development. In the contracting environment the customer sets many more formal demands on the contractor's 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 between the modules. Simultaneously developers engage in articulation work by negotiating and coordinating their efforts with others working 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. 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 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 implication implications. As my date 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 developer 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 component 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 codes that the other has worked on align their changes.

In Tool Corporations 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. In Computer Corporations 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 apiece 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. In Tool Corporations developers either know who the experts are or use the tool to find out who has been working on code in that sub-system. In Computer Corporations 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, 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.

In Tool Corporations people use the organizational memory to help them learn from the past, and coordinate their efforts with the echoes of previous developers. In Computer Corporations the developers also use 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 do. Contract Corporations developers also manage 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 o n 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 wons 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 time sonly 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 revise code, the modules that fix the problem or deliver the required enhancement.

In Tool Corporations the developers learn 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. In Computer Corporations 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 Corporation’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 do not end up being the system that they have to integrate their fix with. In Tool Corporations the system could evolve in 14 different ways simultaneously, as there were 14 developers there. In Computer Corporations 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 software 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. In Computer Corporations integration happens on two levels. Building sub-systems and putting the whole system together. Gathering the –sub-system in Compute Corporations approximates the same effort required by Tool Corporations 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). Developers integrating systems observe another technical aspect of integration dependencies, aligning current versions of all the modules. This involves gathering all the most recent version of the components to go into the final system, and ensuring that either all of a change gets in, or stays out.

Both Tool Corporation and Computer Corporations have developers assigned to the role of build manager, the person responsible for collecting the newest changes of the system and putting them together. In Tool Corporations 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 o f much of the work involved in finding the most recent changes and ordering the technical executing 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. In Computer Corporations the build manger 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 Corporations also reveal how as software grows in size the complexity of managing integration dependencies rise. 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 Corporations have job function, builder, whose job consists of doing nothing but

integrating large systems. Computer Corporations also have to integrate the entire system. They have an organizational division, the release group, who perform that function.

Practice 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. In Tool Corporations 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 Corporation rely on the tool implicitly and make assumption s about what other people are doing based on information present in the system. However, it was the developers at Computer Corporations who really reveal the extent to which a tool can shape the way that these dependencies are coped with. For example, the shift from Alpha to Tool Corporation'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 back track using the tool, others changed, and often the developers at Computer Corporation want help in making the cognitive shift between the old and new tool ways of working.

In Contract Corporations the configuration managers do all of the work that the tools do in Tool Corporations and Computer Corporations The configuration manager also had the responsibility of making the adjustments as tools changed. 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 were not the development environment tools but the document production systems. The differences created by the configuration management demands of non-commercial 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 Corporations want 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.

Platforms 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. Adaptations of practice do not only occur when the organization changes tools. In Computer Corporation it also happened when the organization expanded its product lint to compete in new markets. As well as managing the tow 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.

7.3Group 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 Corporations and Computer Corporations 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. Establish this big picture was something that groups with in both organizations struggled with. In Tool Corporations 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. In Computer Corporations the developers realized that they would not be able to visualize the system as a whole, but they still wanted to understand their sub-stem and its relationships with other parts of the product. Some people, architects, serves as boundary spanners for the developers at Computer Corporations 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 no only how the system fits together, but how I interact as a whole was especially apparent in Computer Corporations 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 Corporations calls shared code: libraries and functions that many sub-systems rely on. They also require management, and Computer Corporations deal 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 Corporations 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. In Tool Corporations they develop a point release (an upgrade for the existing customer base) and a new product simultaneously. These companies also develop each product on multiple platforms, and some of the platform developments took longer than others. The developers are divided into groups working on distinct platforms, and although they want to maintain consistent functionality across the platforms they have to separate the life cycles because they need to go at different development speeds. Technically, the speed issue manifested itself during the nightly builds. The newer platform code is not ready to be built each night because it breaks the build, consequently impacting the work of the developers working on the old platform. This means often both platforms need to be compared to see whether they provide consistent functionality.

In Computer Corporations 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 toward 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 version 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 Corporations provide 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 Corporations and Computer Corporations 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 Corporations or computer Corporations must revise their development schedule to take account of the new release. Usually this means caring 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 Corporations and Computer Corporations 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 do no 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 Corporations and Computer Corporations 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. They also depend on the organizations for support and on-going compatibility. Organizations also use focus groups,

where customers provide feedback on ht 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 identifies: the need to integrate the system, the on-going changes, and the existence of multiple product's and external demands on the development process. However, these causes have intricate relationships. The connections between the causes are described in this section.

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 t impacting the life-cycle, external demands also shapes 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 these assemblies distinct even though one code module may fit into more than one product.

Finally systems evolutions, the on-going changes to the product, imply that integration must occur more that 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 place don 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: 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 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 face 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 decomposition, 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 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, and 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 ad further complexity to these dependencies all development organizations find themselves in a social world with conventions that 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.

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 the 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 with in 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)

Parnas clearly believes that good decomposition would eliminate the complexity of recompositions.

The data presented in this thesis challenges this assertion. Clearly the three organizations studied do not have the managerial, product flexibility, or comprehensibility gains that Parnas describes. This raises two questions: do Parnas get this wrong or do the organizations 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 coed. 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 up 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 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 kind's o f 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 do 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 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 of span functional divisions. However, some of the individual dependencies also span groups. When special builders take on the build management role for a tram 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 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

Babich, W.A. (1986). Software configuration management: Co-ordination for team productivity.Reading MA:Addison-Wesley.

Bassili,V.R. and Musa J.D.(1991). The future engineering of software: A management perspective.IEEE computer, vol.24,no. 9,90-96.

Bendifallah,S., and Scacchi,W.(1987). Understanding software mantanace work.IEEE transaction on software engg.,vol.10,no,1,79-87

Boehm,B.W.(1981) sftware engg. 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.

Brooks Jr,F.P.(1975) . The Mythical Man-Month:Essays on software engg. Reading,MA:Adisson-Wesley.

Brooks Jr,F.P.(1987). No Silver Bullet: Essence and Accidents of Software Engg. IEEE COMPUTER , Vol. 20, No. 4,10-19.

H. Dieter Rombach, Victor R. **Basili**, Richard W ... - 1993 - Computers - 261 pages *Basili VR, Musa JD*: The Future Engineering of Software: A Management Perspective. IEEE

Brooks Jr,F.P.(1995). The Mythical Man-Month: Essays on software engg. (20th Anniversary Edition ed.) Reading , MA: Addison-Wesley.

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.

Davies, L., and Nielsen, S. (1992) . An Ethnographics 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 System Development, 179-192. Amsterdam: Elsevier Science Publishers B.V.

Gamma, E., Helm, R., and Star, S.L. (1986). Analyzing Due Process in the Workplace. ACM Transactions on Office System , vol. 4, no. 3, 257-270

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). Fundamentals of Software Engg. Englewood Cliffs, N.J.: Prentice-Hall.

Babich, *W.A.* (*1986*). Software Configuration Management, Coordination for Team Productivity. 1st edition. Boston: Addison-Wesley; Berczuk, Appleton; (2003).

Babich, *W.A.* (*1986*). Software Configuration Management, Coordination for Team Productivity. 1st edition. Boston: Addison-Wesley; Berczuk, Appleton; (2003).

H. Dieter Rombach, Victor R. **Basili**, Richard W ... - 1993 - Computers - 261 pages *Basili VR*, *Musa JD*: The Future Engineering of Software: A Management Perspective. IEEE

By MCataldo2009 Recent *studies* suggest, however, that the identification and *management*. What is the *relative* impact of syntactic and logical *dependencies* **.** We examined our *research* questions using two large *software development* projects.

By MCatald-2008 evolution literature has introduced a *new* view on technical *dependencies* **.** contribution of this *study* is the examination of the *relative* impact **.** We examined our *research* questions using two large *software development* projects.