A Coordination Theory Approach to Organizational Process Design

Kevin Crowston
Syracuse University, School of Information Studies, Syracuse, New York 13244-4100

Design of new organizational forms can begin with an analysis of existing organizational processes and identification of ways to change these process arrangements. Kevin Crowston applies coordination theory to show how task processes can be decomposed, documented, and altered to create new forms of organizing work. Crowston's research demonstrates the potential of coordination theory in the study of new organizational forms and process redesign.

Gerardine DeSanctis

Abstract

An important practical problem for many managers is finding alternative processes for performing a desired task, for example, one that is more efficient, cheaper, or that is automated or enhanced by the use of information technology. Improving processes also poses theoretical challenges. Coordination theory provides an approach to the study of processes. In this view, the design of a process depends on the coordination mechanisms chosen to manage dependencies among tasks and resources involved in the process.

In this paper, I use coordination theory to analyze the software change process of a large mini-computer manufacturer. Mechanisms analyzed include those for task assignment, resource sharing, and managing dependencies between modules of source code. For each, I suggest alternative mechanisms and thus alternative designs for the process. The organization assigned problem reports to engineers based on the module that appeared to be in error, since engineers only worked on particular modules. Alternative task assignment mechanisms include assignment to engineers based on workload or market-like bids. Modules of source code were not shared, but rather "owned" by one engineer, thus reducing the need for coordination. An alternative resource sharing mechanism would be needed to manage source code if multiple engineers could work on the same modules. Finally, engineers managed dependencies between modules informally, relying on their personal knowledge of which other engineers used their code; alternatives include formally defining the interfaces between modules and tracking their users.

Software bug fixing provides a microcosm of coordination problems and solutions. Similar coordination problems arise in most processes and are managed by a similar range of mechanisms. For example, diagnosing bug reports and assigning them to engineers may have interesting parallels to diagnosing patients and assigning them to specialists.

While the case presented does not formally test coordination theory, it does illustrate the potential of coordination theory for exploring the space of organizational processes. Future work includes developing more rigorous techniques for such analyses, applying the techniques to a broader range of processes, identifying additional coordination problems and mechanisms and developing tools for collecting and comparing processes and automatically suggesting potential alternatives.

(*Organization Theory*; *Coordination Theory*; *Organizational Processes*; *Process Redesign*; *Process Reengineering*)

1. Introduction

Describing and categorizing organizational forms remains a central problem in organization theory (McKelvey 1982, Rich 1992, Sanchez 1993). Unfortunately, defining organizational form poses numerous difficulties. Mohr (1982) describes organizational structure as "multidimensional—too inclusive to have constant meaning and therefore to serve as a good theoretical construct." McKelvey and Aldrich (1983) point out that most large organizations are actually mixtures of different forms. In other words, an entire organization

1047-7039/97/0802/0157/$05.00

is too aggregate a level of analysis for meaningful comparison of forms.

To narrow the study of organizational forms, some researchers have suggested focusing on how particular tasks are performed, using the process as the focus of analysis (Mohr 1982, Abbott 1992). For example, to understand how General Motors and Ford are alike or different, researchers might compare their automobile design processes or even more specific subprocesses. The problem thus becomes not what structural form an organization has, but what process it uses to accomplish a particular task.

Given this perspective, an important practical problem is to identify alternative processes that would also be suitable for performing a desired task. As companies scramble to adapt to increasingly frequent environmental changes, this question has become even more pressing. For example, although managers may realize that the survival of their company depends on reducing time-to-market and improving quality, they may find it difficult to translate these goals into concrete organizational changes, e.g., as part of a business process redesign effort (Davenport and Short 1990, Hammer 1990, Harrison and Pratt 1993). Other managers may be concerned with making effective use of information technology (IT), electronic media in particular, and wonder what kinds of organizational processes become possible as the historic constraints on communications and information processing are relaxed. Underlying both these questions is the central theoretical issue: how can we represent organizational processes in a way that allows us to compare and contrast them or to design new ones (Malone et al. 1993)?

Consider the software problem (bug) fixing process, a process that I studied and which I will use as a source for examples in this paper. Customers having problems with a piece of software report the problems to its developers, who (they hope) eventually provide some kind of solution. The company I studied, the developer of a mini-computer operating system, has an elaborate process to receive problem reports, filter out duplicates of known problems, identify for novel problems which modules of the system are apparently at fault and route the reports to the software engineers responsible for those modules. Along the way, an engineer might develop a workaround (i.e., a way to avoid the problem); the responsible software engineer might develop a patch (i.e., a change to the code of part of the system) to fix it. The patch is then sent to other groups who test it, integrate it into the total system and, eventually, send it to the customers who originally had the problem. (A more detailed description of this process appears in §3.) This analysis raises several questions. Why is the process structured this way, with finely divided responsibility for different parts of the process? More simply, how else could the company do this?

In the remainder of this paper, I present one approach to answering these questions. In the next section I briefly review coordination theory and show how it can guide the analysis and redesign of a process. The bulk of the paper presents a detailed example. Section 3 describes the case site—the software development division of a minicomputer manufacturer—and the data collection method. Section 4 discusses the dependencies and coordination mechanisms identified in the case, and suggests possible alternative mechanisms and therefore processes. The paper concludes by briefly evaluating the coordination theory approach and discussing its application in other settings.

2. A Coordination Theory Approach to Organizational Processes

In this paper, I use coordination theory as one approach to analyzing and redesigning processes. If we examine many companies, we will observe a wide variety of approaches to the software bug fixing process. For example, in other companies (and other parts of the company I studied), when a problem report arrives, it is simply assigned to the next free engineer. If we examine many processes, we will see a similar range of possibilities. Individuals (or firms) may be either generalists who perform a wide variety of tasks, or specialists who perform only a few. Activities may be assigned to actors within a single organization, as with bug fixing; other assignments may take place in a market, as with auditing, consulting and an increasingly wide variety of services; and finally, assignments may be given to others in a network of corporations (Powell 1990).

Despite this diversity, when we systematically compare processes, patterns emerge. Organizations that perform the same task often perform essentially the same basic activities. For example, organizations that fix software bugs must all diagnose the bug, write code for a fix, and integrate the change with the rest of the system. Looking more broadly, many engineering change processes have activities similar to those for software.

While these general activities are often the same, the processes differ in important details: how these large abstract tasks are decomposed into activities, who performs particular activities, and how they are assigned. In other words, processes differ in how they are coordinated. However, even with coordination there are common patterns: similar problems arise and are managed similarly. For example, nearly every organization must assign activities to specific actors and task assignment mechanisms can be grouped into a few broadly similar categories. Such mechanisms are the subject matter of coordination theory.

To analyze these patterns of coordination, I use the framework developed by Malone and Crowston (1994), who define coordination as "managing dependencies between activities" (p. 90). They define coordination theory as the still-developing body of "theories about how coordination can occur in diverse kinds of systems" (p. 87). Malone and Crowston analyze group action in terms of *actors* performing *interdependent activities* to achieve *goals*. These activities may also require or create *resources* of various types.

For example, in the case of software bug fixing, *activities* include diagnosing the bug, writing code for a fix, and integrating it with the rest of the system, as mentioned above. *Actors* include the customers and various employees of the software company. In some cases, it may be useful to analyze a group of individuals as a collective actor (Abell 1987). For example, to simplify the analysis of coordination within a particular subunit, the other subunits with which it interacts might all be represented as collective actors. The *goal* of software bug fixing appears to be eliminating problems in the system, but alternative goals—such as appearing responsive to customer requests—could also be analyzed. In taking this approach, we adopt Dennett's (1987) intentional stance: because there is no completely reliable way to determine someone's goals (or if indeed they have goals at all), we, as observers, can only impute goals to the actors and analyze how well the process accomplishes these goals. Finally, *resources* include the problem reports, information about known problems, computer time, software patches, source code, and so on.

According to coordination theory, actors in organizations face *coordination problems* that arise from dependencies that constrain how tasks can be performed. These dependencies may be inherent in the structure of the problem (e.g., components of a system may interact with each other, constraining the kinds of changes that can be made to a single component without interfering with the functioning of others) or they may result from decomposition of the goal into activities or the assignment of activities to actors and resources (e.g., two engineers working on the same component face constraints on the kind of changes they can make without interfering with each other).

To overcome these coordination problems, actors must perform additional activities, which compose what Malone and Crowston call *coordination mechanisms*. For example, a software engineer planning to change one module in a computer system must first check if the changes will affect other modules and then arrange for any necessary changes to modules that will be affected; two engineers working on the same module each must be careful not to overwrite the other's changes. Coordination mechanisms may be specific to a particular setting, such as a code management system to control changes to software, or general, such as hierarchical or market mechanisms to manage assignment of activities to actors or other resources.

The first key claim of coordination theory is that dependencies and the mechanisms for managing them are general, that is, a particular dependency and a mechanism to manage it will be found in a variety of organizational settings. For example, a common coordination problem is that a particular activity may require specialized skills, thus constraining which actors can work on it. This dependency between an activity and an actor arises in some form in nearly every organization. Coordination theory thus suggests identifying and studying common dependencies and their related coordination mechanisms across a wide variety of organizational settings.

The second claim is that there are often several coordination mechanisms that could be used to manage a dependency, as the task assignment example illustrates. Possible mechanisms to manage the dependency between an activity and an actor include manager selection of a subordinate, first-come-first-served allocation and various kinds of markets. Again, coordination theory suggests that these mechanisms may be useful in a wide variety of organizational settings. Organizations with similar activities to achieve similar goals will have to manage the same dependencies, but may choose different coordination mechanisms, thus resulting in different processes.

Finally, the previous two claims taken together imply that, given an organization performing some task, one way to generate alternative processes is to first identify the particular dependencies and coordination problems faced by that organization and then consider what alternative coordination mechanisms could be used to manage them.

To summarize, according to coordination theory, the activities in a process can be separated into those that are necessary to achieve the goal of the process (e.g., that directly contribute to the output of the process) and those that serve primarily to manage various dependencies between activities and resources. This conceptual separation is useful because it focuses attention on the coordination mechanisms, which are believed to be a particularly variable part of a process, thus suggesting an approach to redesigning processes. Furthermore, coordination mechanisms are primarily information-processing activities and therefore, good candidates for support from information-technology.

More broadly, coordination theory provides a way to study the development of new organizational forms. Form includes structure but also process, so changes in process directly affect organizational form. A core aspect of process is the design of individual tasks, and coordination is at the root of task design, as tasks are decomposed and assigned to multiple actors. Therefore, the study of coordination mechanisms and the coordination of organizational processes provides a fundamental way to analyze existing and developing new organizational forms.

The aim of coordination theory is not new: defining processes and attempting to improve performance has been a constant goal or organization theory. The focus on dependencies is also a recurring theme. Even the idea of substitute mechanisms has been suggested; for example, Lawler (1989) argues that the functions of an organization's hierarchy, many of which are ways of coordinating lower level actions, can be accomplished in other ways, such as work design, information systems or new patterns of information distribution. However, coordination theory makes many of these earlier notions more precise by decomposing tasks and resources. For example, the classic distinction among sequential, interdependent, and network processes of organizing can be decomposed into particular dependencies managed by particular mechanisms. In this view, a network, for example, is not a property of a collection of organizations *per se*, but rather a restriction on which actor is chosen to work on a particular task (i.e., how a task-actor dependency is managed). In a hierarchy, a task is assigned to an actor chosen from within the organization, e.g., based on specialization or managerial decision; in a market, from the set of suppliers active in the market, e.g., by bidding; and in a network, from the appropriate member of the network.

2.1. A Typology of Coordination Mechanisms

As a guide to such analyses, Crowston (1991) presents a preliminary typology of dependencies and associated coordination mechanisms. Refining and completing this typology is an important ongoing research project, but a preliminary typology of dependencies and examples of associated coordination mechanisms is shown in Table 1.

The main dimension of the preliminary typology involves the types of objects involved in the dependency. To simplify the typology, we compress the elements of Malone and Crowston's (Malone and Crowston 1994) framework into two groups: tasks (which includes goals and activities) and resources used or created by tasks (which here includes the effort of the actors). Logically, there are three kinds of dependencies between tasks and resources: those between two tasks, those between two resources and those between a task and a resource. (As a further simplification, dependencies between more than two elements will be

Table 1 A Typology of Dependencies and Associated Coordination Mechanisms from Crowston (1991)

Dependency	Coordination Mechanisms to Manage Dependency
Task-task	
Tasks share common output	
same characteristics	1 look for duplicate tasks
	2 merge tasks or pick one to do
overlapping	• negotiate a mutually agreeable result
conflicting	• pick one task to do
Tasks share common input (shared resource)	
shareable resource	• no conflict
reusable resource	1 notice conflict
	2 schedule use of the resource
nonreusable resource	• pick one task to do
Output of one task is input of other (prerequisite)	
same characteristics	1 order tasks
	2 ensure usability of output
	3 manage transfer or resources
conflicting	• reorder tasks to avoid conflict
	• add another task to repair conflict
Task-resource	
Resource required by task	1 identify necessary resources
	2 identify available resources
	3 choose a particular set of resources
	4 assign the resources
Resource-resource	
One resource depends on another	1 identify the dependency
	2 manage the dependency

decomposed into dependencies between pairs of elements.)

Some cells of this typology are more developed than others; for example, task-task dependencies have been analyzed in some detail, while the others have not. Task-task dependencies are distinguished by considering what kinds of resources are shared by the two tasks (e.g., shareable or reusable), how these resources are used (as an input or as an output of the task), and whether the required uses conflict with each other.

For each dependency, a brief description of an associated coordination mechanism is given. For example, to manage a task-resource dependency, the typology notes that it is necessary first to identify required and available resources, then to choose a particular resource and finally to assign the resource. Managing a prerequisite dependency (a task-task dependency) requires ordering the tasks, ensuring that the output of the first is usable by the second and managing the transfer of the resource from the first to the second. These activities can be performed in many different ways. For example, a manager with a task to assign might know of the available resources or might have to spend time hunting them down. Usability might be managed reactively by testing the resource and returning problems or proactively by involving the user in the production of the resource.

3. Data and Methods

Coordination theory is intended to analyze organizations in a way that facilitates redesign. The question is, does this approach work, that is, can we find dependencies and coordination mechanisms in a real process? Does this analysis help explain commonly used alternative processes or suggest novel ones? I undertook a case analysis to answer these questions in a setting where the precise processes for decomposing and completing tasks were observable. In the remainder of this paper, I present the application of coordination theory to a particular process, thus grounding the claims of coordination theory within a carefully specified organizational domain.

3.1. Research Setting

In the remainder of this paper, I examine a software change process, considering alternative forms the process could take. The organization in this example was the minicomputer division of a large corporation. In 1989, when the study started, the entire corporation had sales of approximately $10 billion and roughly 100,000 employees. The computer division produced several lines of minicomputers and workstations and developed system software for these computers.

Engineering Change Processes. This process was examined as part of a larger study of engineering change processes. Engineering change processes are interesting for several reasons. First, these processes require individuals in engineering and manufacturing to coordinate to be effective. Second, even though each change is unique, the management of these changes is routine. Pentland (1992) notes advantages to studying this sort of routine work: each change forms a clearly bounded unit of work with intensive records kept of each one. Finally, nearly all manufacturing companies must manage product changes, so ideas for improving the performance of this process could be of great value to such organizations.

Although three organizations were included in the full study, this paper focuses on the change process in a software development organization. Software is interestingly different from other products because it is not a physical product. This product nature has two implications. First, the rate of changes is higher in software than in hardware. Second, software problems are more likely to be systematic than are hardware problems. A problem with a piece of hardware may or may not occur in another item: the problem may be due to a design flaw, but it may also be a random failure. On the other hand, a problem with a piece of software is likely to occur in every copy of the software. As a result, it is particularly important that software changes be carefully controlled.

Lientz and Swanson (1980) distinguish three reasons for making changes to a program: corrective, perfective, and adaptive. Corrective changes are those made to fix problems. For the company in this study, problems are defined as disagreements between the behaviour of a program and its documentation. Fixing problems usually requires changing the software to make it agree with the documentation, but sometimes the fix is made to the documentation instead. Perfective changes are those made to improve a correct program without altering its behaviour (e.g., to improve performance). Finally, adaptive changes are those that add new functionality, altering the software to meet changing requirements. The last two categories of changes are system enhancements rather than bug fixes and will not be discussed further in this paper.

Operating System Development. The group in this study was responsible for the development of the kernel of a proprietary operating system, a total of about one million lines of code in a high-level language. An operating system is the basic software of the computer; its major function is to insulate programmers from the details of the hardware. Additional functions permit multiple users to share the computer without interference. Increasingly, operating systems provide specialized services such as access to a network or database and transaction management. The operating system in this case study was broken into several subsystems, such as the process manager or file system; each subsystem was further divided into modules, each of which implements a small set of services.

3.2. Data Collection

The analysis presented here was based on 16 interviews with 12 individuals, including six software engineers, two support group managers and three support group members and one marketing engineer. The interviews were carried out during six trips to the company's engineering headquarters; most were one to two hours long. Additionally, a former member of the software development group assisted in the data collection and analysis.

As discussed above, coordination mechanisms are primarily information-processing activities. Therefore, this study adopted the information processing view of organizations, which focuses on how organizations process information (March and Simon 1958, Galbraith 1977, Tushman and Nadler 1978). The goal of the data collection was to uncover, in March and Simon's (1958) terms, the programs used by the individuals in the group. March and Simon suggest three ways to uncover these programs: (1) interviewing individuals, (2) examining documents that describe standard operating procedures, and (3) observing individuals. I relied most heavily on interviews. As March and Simon (1958) point out, "most programs are stored in the minds of the employees who carry them out, or in the minds of their superiors, subordinates, or associates. For many purposes, the simplest and most accurate way to discover what a person does is to ask him" (p. 142).

I started the data collection by identifying different kinds of actors in the group. This identification was done with the aid of a few key informants, and refined as the study progressed. When available, formal documentation of the process was used as a starting point. For example, a number of individuals designed and coded parts of the operating system, all working in roughly the same way and using the same kinds of information; each was an example of a "software engineer actor." However, response centre or marketing engineers used different information, which they process differently. Therefore they were analyzed separately.

Interview subjects were identified by the key information based on their job responsibilities; there was no evidence, however, that their reports were atypical. I then interviewed each subject to identify the type of information received by each kind of actor and the way each type was handled. Data were collected by asking subjects: (1) what kinds of information they received; (2) from whom they received it; (3) how they received it (e.g., from telephone calls, memos or computer systems); (4) how they processed the different kinds of information; and (5) to whom they sent messages as a result. When possible, these questions were grounded by asking interviewees to talk about items they had received that day.

I also collected examples of documents that were created and exchanged as part of the process or that described standard procedures or individual jobs. Not surprisingly, the process as performed often differs from the formally documented process. For example, there was a formal method for tracking which engineers used which interfaces, but in practice most engineers seemed to rely on their memories. It was this informal process (as well as the formal process surrounding it) that I sought to document.

3.3. Validation of Data

The initial product of these studies was a model of the change process (presented in more detail below) that described the actors involved, which steps each performed and the information they exchanged. It should be noted that data were collected about only one group because my contact at this company worked in that group. My impression from interviews with individuals who had worked in or who interacted with other groups, was that processes were similar in other software development units, however, I had no direct information about other groups.

Relying on interviews for data can introduce some biases. First, people do not always say what they really think. Some interviews were conducted in the presence of another employee of the company, so interviewees may have been tempted to say what they think they should say (the "company line"), what they think I want to hear or what will make themselves or the company look best. Second, individuals sometimes may not know the answer or may be mistaken.

To control for interview bias, I cross-checked reported data with other informants. I also used the modelling process as a check on the data, applying the negative case study method (Kidder 1981). In this method, researchers switch between data collection and model development, using predictions or implications of the model to guide the search for disconfirming evidence. When such data can not be found, the model has been refined to agree with all available data.

3.4. The change process

Goals of the Change Process. The organization stated the following goals for the change process:

• ensure that all critical program parameters are documented: customer commitments, cross-functional dependencies.

• ensure that a proposed change is: reviewed by all impacted software development units/functions and formally approved or rejected.

• ensure that document status is made available to all users: stable (revision number and date); changes being considered; approved/rejected/withdrawn.

• ensure changes are made quickly and efficiently.

In addition, the change process has two larger goals: maintain the quality of the software and to minimize the cost of changes. To maintain quality, the process ensures that changes are made by someone who understands the module involved, that changes are fully tested, and that the module and its documentation are kept in agreement. To reduce the cost of changes, the change process requires that changes be made only to fix a problem or add an authorized enhancement. As one manager put it, the "formal change control process is there to prevent changes."

The activities performed for a typical change are summarized in the flowchart shown in Figure 1. Actors involved in the process are listed at the top of the column of activities they perform. To save space, the flow continues from the bottom right of the chart to the top left; the activities on the right follow rather than overlap those on the left. A short description of the steps in this process is given in Appendix A and a more detailed description is available from the author. (Note that the response centre was treated as a collective actor. As a result, internal centre activities are omitted from the flowchart of the process.) Although no particular bug is necessarily treated in exactly this way, these activities were described as typical by my interviewees.

4. Dependencies and Coordination Mechanisms in Software Bug Fixing

Given a process, the claim of coordination theory is that we can generate alternative processes by first identifying the coordination mechanisms currently in use and then trying alternative coordination mechanisms in their place. In the following two sections, I first describe the current coordination mechanisms and then discuss the implications of various alternative mechanisms.

4.1. Identifying Current Dependencies and Coordination Mechanisms

While making a change to the system, numerous dependencies must be managed. The best way to identify dependencies and coordination mechanisms is still an active area of research, but three heuristics seem plausible (Osborn 1993). First, we can examine activities in the current process identify those that seem to be part of some coordination mechanism, and then determine what dependencies they manage. Second, we can list the activities and resources involved in the process, consider what dependencies are possible between them, and then determine how these dependencies are being managed. Finally, we can look for problems with the process that hint at unmanaged coordination problems and identify the underlying dependencies.

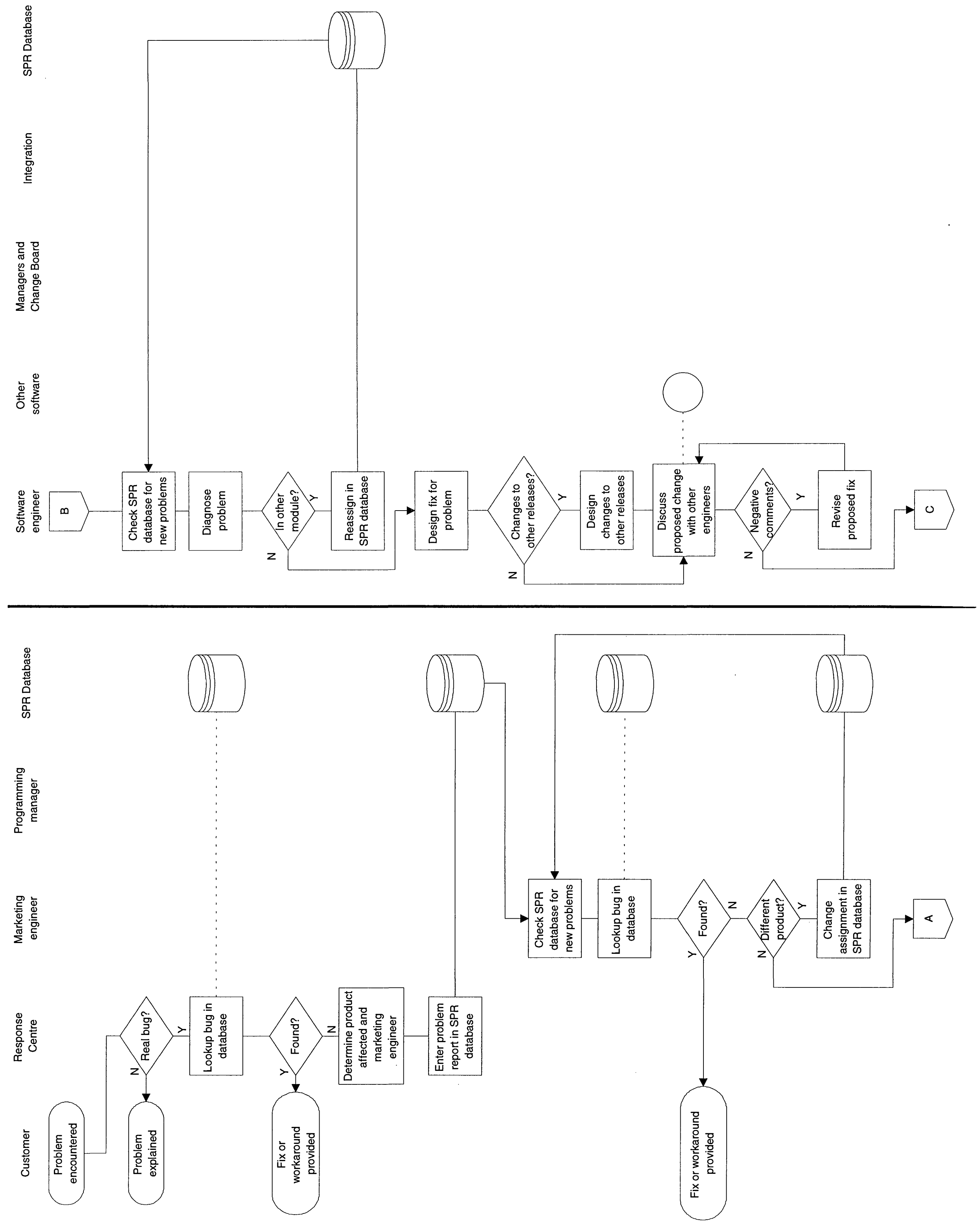
4.1a. Looking for Coordination Mechanisms

Taking the first approach, many of the activities in the bug fixing process appear to be instances of the coordination mechanisms discussed earlier. Table 2 lists the activities performed in the current bug fixing process. The dependency the activity manages, if any, is listed in the third column.

For example, one of the first things the customer service centre staff and marketing engineers do upon receiving a problem report is check if it duplicates a known problem listed in the Software Problem Report (SPR) database. In the typology, *looking for duplicate tasks* is listed as a coordination mechanism for managing a dependency between two tasks that have duplicate outcomes. The organization can avoid doing the same work twice by noticing the duplication and reusing the result of one of the tasks (as happened in this example).

Task assignment is a coordination mechanism for managing the dependency between a task and an actor by finding the appropriate actor to perform the task. Such coordination mechanisms are performed repeatedly in this process: customers assign tasks to the customer service centre, the customer service centre assigns novel tasks to the marketing engineers, marketing engineers assign them to the software engineers and software engineers assign tasks to each other.

Figure 1. **Flowchart of software bug fixing process.**



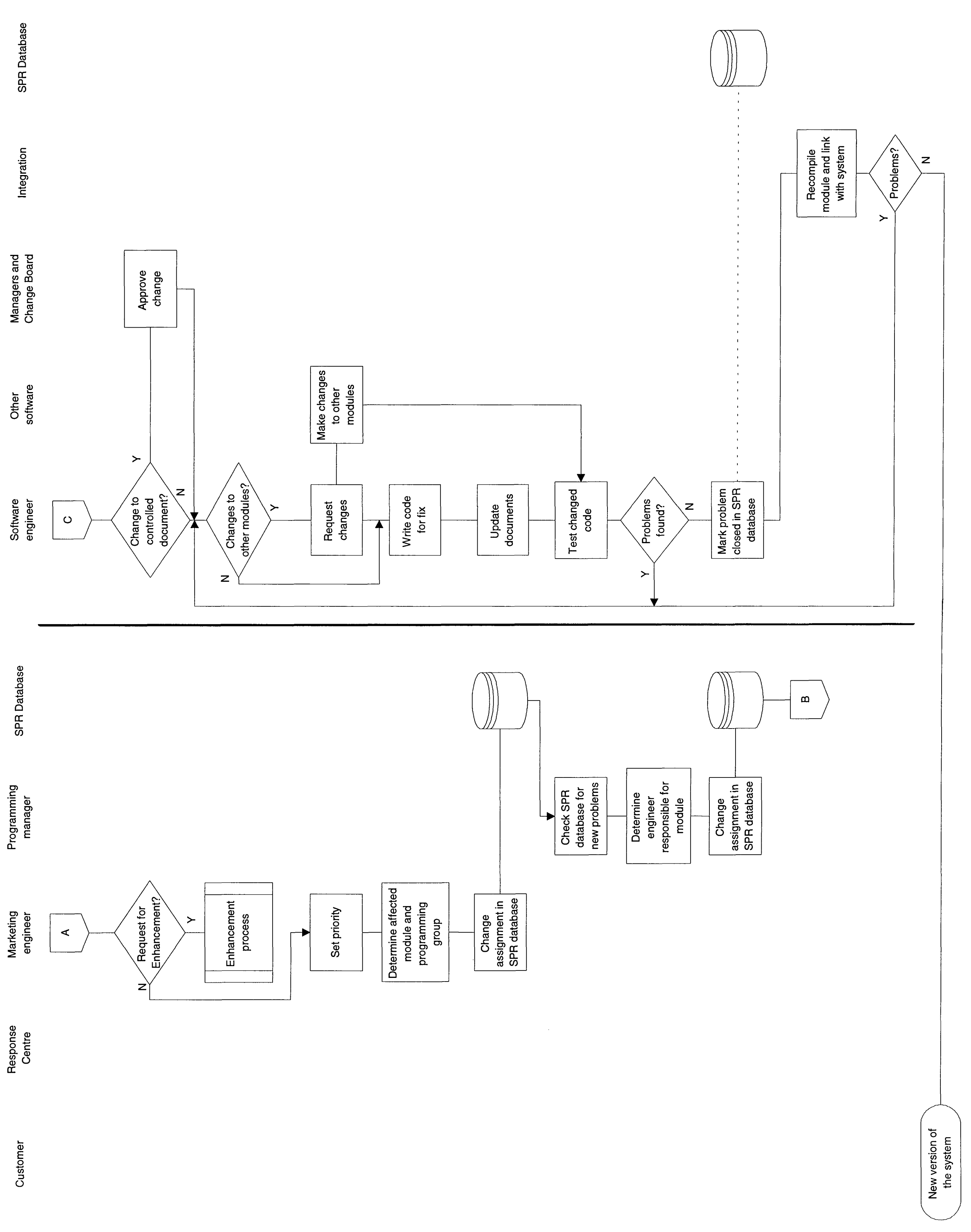

Customer
Response Centre
Marketing engineer
Programming manager
SPR Database
A
Request for Enhancement?
Y
N
Enhancement process
Set priority
Determine affected module and programming group
Change assignment in SPR database
Check SPR database for new problems
Determine engineer responsible for module
Change assignment in SPR database
B
Software engineer
Other software
Managers and Change Board
Integration
SPR Database
C
Change to controlled document?
Y
N
Approve change
Changes to other modules?
Y
N
Request changes
Make changes to other modules
Write code for fix
Update documents
Test changed code
Problems found?
Y
N
Mark problem closed in SPR database
Recompile module and link with system
Problems?
Y
N
New version of the system

Table 2 Activities in the Software Problem Fixing Process

Actor	Activity	Dependency Managed between . . .
Customer	**Find a problem while using system**	
	Report problem to response centre	Problem fixing task and capable actor
Response Centre	Look for bug in database of known bugs; if found, return fix to customer and stop	Problem fixing task and duplicate tasks
	Attempt to resolve problem	
	Refer hardware problems to field engineers	Problem fixing task and capable actor
	If problem is novel, determine affected product and forward bug report to marketing engineer	Problem fixing task and capable actor
Marketing Engineer	Look for bug in database of known bugs; if found, return fix to customer	Problem fixing task and duplicate tasks
	Request additional information if necessary	Usability of problem report by next activity
	Attempt to reproduce problem	Usability of problem report by next activity
	Attempt to find workaround	
	Set priority for problem	Problem fixing task and actor's time
	If the report is actually a request for an enhancement, then treat it differently	
	Determine affected module	Task and resources required by tasks
	If unable to diagnose, forward to SWAT Team	
	If bug is in another product, forward to appropriate product manager	Problem fixing task and capable actor
	Forward bug report to manager of group responsible for module	
Programming Manager or Designate	Determine engineer responsible for module and forward bug report to that engineer	Problem fixing task and capable actor
Software Engineer	Pick the report with the highest priority, or the oldest, or the one you want to work on next	Problem fixing task and actor's time
	Diagnose the problem	
	If the problem is in another module, forward it to the engineer for that module	Problem fixing task and capable actor
	Design a fix for the bug	
	Check if change is needed in other releases and make the change as needed	
	Send the proposed fix to affected engineers for their comments; if the comments are negative, then revise the bug and repeat the process	Two modules
	If the change requires changes to a controlled document, then send the proposed change to the various managers and the change review board for their approval	Management of usability task and capable actor
Managers	Approve the change	Usability of fix by next activity
Software Engineer	**Write the code for the fix**	
	Determine what changes are needed to other modules	Task and subtasks needed to accomplish it
	If necessary, ask the engineers responsible for the other modules to make any necessary changes	Problem fixing task and capable actor
	Test the proposed fix	Usability of fix by next activity
	Send the changed modules to the integration manager	Task and capable actor
	Release the patch to be sent to the customer	Transfer to customer
Integration	Check that the change has been approved	Usability of fix by integration activity
	Recompile the module and link it with the rest of the system	
	Test the entire system	Usability of entire system by next activity
	Release the new software	Transfer to customers

4.1b Looking for Dependencies

The second approach to identifying coordination mechanisms is to list the tasks and resources involved in the process and then consider what dependencies are possible between them. It may be that some of the activities in a process are coordination mechanisms for managing those dependencies. As mentioned above, tasks necessary to respond to problem reports include noticing there is a problem, finding a workaround, reproducing and diagnosing the problem, designing a fix, writing new code and recompiling the system with the new code. These activities are shown in bold in Table 2. Resources (in the sense of the typology of coordination mechanisms) include the problem reports, the efforts of a number of specialized actors, and the code itself.

Task-task Dependencies. Dependencies between tasks can be identified by looking for resources used by more than one task. For example, many tasks create some output, such as a bug report, a diagnosis, or new code, that is used as input by some other task, thus creating a *prerequisite dependency* between the two. Malone and Crowston (1994) note that such dependencies often impose usability and transfer constraints. Some steps in the process appear to manage such constraints. For example, testing that a new module works correctly addresses the usability constraint between creating code and relinking and using the system; releasing the new system addresses the transfer constraint between the company and the final user of the system.

If there are two problems in the same module, then both bug fixing tasks need the same code, thus creating a *shared resource dependency.* In this process, this dependency is managed by assigning modules of code to individual programmers and then assigning all problems in these modules to that programmer. This arrangement is often called "code ownership," because each module of the system has a single owner who performs all tasks that modify that module. Such an arrangement allows the owner of the code to control all changes made to the code, simplifying the coordination of multiple changes.

Task-resource Dependencies. The second category of dependencies is those between tasks and resources, which are managed by some kind of task or resource assignment. These coordination mechanisms were identified and discussed above.

Resource-resource Dependencies. Finally, there are dependencies between modules owned by different engineers, that is, resource-resource dependencies, that constrain what changes can be made. A module depends on another if the first makes use of services provided by the second. For example, the process manager may use routines that are part of the file system; therefore, the process management code depends on the file system code.

Such dependencies must be noticed or identified before they can be managed by arranging for coordinated changes. Interactions between different parts of a software system are not always obvious, since they are not limited to direct physical connections. In principle, it should be easy to detect dependencies automatically by examining the code. In practice, however, there seem to be no reliable mechanical means to determine the interactions between different modules.

Instead, dependencies are tracked by manually tracking documents. The set of routines and data provided by a module make up what is called the interface to that module. Different interfaces are provided for different classes of users. Customer interfaces are described in published manuals and are therefore rarely changed. Service interfaces are provided for use by developers of other parts of the system software and are described in formal documents, called external specifications, which are circulated within the company but usually not to customers. Interfaces intended for use only within a single development group are described in an informally circulated internal specification, if they are documented at all.

Copies of manuals and external specifications are kept in a documentation library; internal specifications are maintained only by their developer. Programmers who request a document from the document library are tracked so they can be informed of any changes to the document. At the time of my study, there were 800 to 900 documents in the library, and about 1000 document requestors being tracked. A total of 15,000 copies of documents had been distributed.

In practice, however, programmers sometimes borrow a document or copy pieces of someone else's code and therefore do not realize that they should inform the developer. Because the documentation lists are not reliable, identification of other affected engineers is done by the software engineer planning a change based mostly on their knowledge of the system's interactions and what other developers are doing.

4.1c Looking for Coordination Problems

A final approach for identifying dependencies is to look for problems in the process that suggest unmanaged dependencies. For example, the company occasionally found at system integration or during testing that a change made to one module was incompatible with others, despite the efforts to detect and avoid interactions described in the previous section. These problems occur because some dependency between the module being changed and other modules were not detected and managed. (In other words, the dependency identified by this approach duplicated one already known.)

Some of these problems can be traced to the heuristic mechanism used to locate dependencies. In particular, because there is little informal communication between divisions, the mechanism does not work very well if modules are developed in different divisions. For example, in the organization studied, the word processing system once became the source of mysterious system crashes. It turned out that the word processor's developers had used a very low-level system call that had been changed between releases of the operating system. However, because the word processor was developed in another unit, the programmers of the two modules did not communicate. Thus, the developers of the word processor did not know they should avoid the system call nor did the developer of the system call know the word processor developers were using it. In other words, the usual social mechanism for finding dependencies between modules failed, leading to the problems.

4.1d Summary

To summarize, there are three heuristics that can be used to identify dependencies and coordination mechanisms in a process. The first approach is to match activities performed against known coordination mechanisms, such as searching for duplicate tasks or task assignment. The second approach is to identify possible dependencies between activities and resources and then search for activities that do manage these. In the example, we identified prerequisite, shared resource, and resource-resource dependencies that were managed. The final approach is to look for problems that suggest unmanaged or incompletely managed dependencies. The coordination mechanisms identified are both generic, such as task assignments, and specific, such as code sharing systems. I believe such analyses will become easier to carry out as we gain experience and build a more complete typology of dependencies and coordination mechanisms.

4.2 Developing New Processes

Given a flowchart of a process such as Figure 1, a common approach to redesign is first to look for problems such as redundant or nonvalue added steps, or places where tasks spend long periods waiting to be worked on, and then to modify the process to address these problems (Harrington 1991, pp. 134–163; Hammer and Champy 1993, pp. 122–126). For example, the current change process assumes that customers cannot fix problems. The process could be modified to allow customers to do more diagnosis, such as checking for known bug fixes. As it happened, a database of documents was developed at the conclusion of our study to allow just this. Customers can dial in to the database and search for documents that describe their problem and the appropriate workaround or patch information. Customers who find solutions can order the patch or apply the workaround; if not, they can leave an electronic request for a return call from the response centre, starting the change process described above.

Coordination theory suggests another approach to redesign, namely, replacing some coordination mechanisms with alternatives. In the remainder of this section, I discuss three examples involving alternative techniques for managing the task-task, resource-resource and task-resource dependencies described above.

Alternative Mechanisms for Managing Prerequisite Dependencies. In problem fixing, several activities ensure that the output of one task is usable by another. For example, marketing engineers check that problem reports are detailed enough to be used by the engineers fixing the bugs; bug fixes are tested at several points to check that they correctly fix the problem and do not introduce new problems.

Along with these tests, managers must approve changes before they can be implemented. Such approvals provide a check on the quality of the change, either directly, if the manager notices problems, or indirectly, if engineers are more careful with changes they show their managers. There are other possible interpretations of this approval process: managers might use the information to allocate resources among different projects, to track how engineers spend their time, or even to demonstrate their political power. However, if approvals are a quality check, other mechanisms might be appropriate, in this and any other process. For example, if approvals are time-consuming yet likely, it may be more effective to continue the change process without waiting for the approval. Most changes will be implemented more quickly; the few that are rejected will require additional rework, but the overall cost might be lower. Alternatively, managerial reviews could be eliminated altogether in favour of more intensive testing and tracking of test results.

Alternative Mechanisms for Managing Resource-resource Dependencies. Resource-resource dependencies are important in complex design processes, particularly in computer software, in which dependencies are not easily visible. Changes with effects outside a single module require coordinated changes to the affected modules. Such dependencies must first be noticed and then managed. However, noticing dependencies can be difficult, as discussed above. One solution to the issue of hidden dependencies is to avoid changes that create problems. This can be done by providing documented service interfaces for the data and calls programmers use, restricting interactions to these interfaces and then freezing the interfaces. A second solution is to try to keep the dependencies visible, in other words, to better track what interfaces people are using. However, the current system does not track all users successfully, and it is unclear what changes would guarantee that all users registered. A final possibility would be to develop mechanisms for finding otherwise hidden dependencies. It seems that this should be straightforward for software, but it was problematic for the organization in this study. However, with more work, such a system could probably be developed.

Alternative Mechanisms for Task Assignment. In the analysis, we noted numerous places where actors perform part of a task assignment process. For example, customers give problem reports to the service centre, which in turn assigns the problems to product engineers, who then assign them to software engineers. In addition, software engineers may assign reports or subtasks to each other.

The typology points out that a key problem in task assignment is choosing the actor to whom to assign a task. Currently, the choice is made based on specialization. This system allows engineers to develop expertise in a few modules, which is particularly important when the engineers are also developing new versions of the system. Furthermore, since modules are assigned to engineers, the code sharing problem discussed above is minimized. However, there are also disadvantages. First, diagnosing the location of a problem can be difficult, because symptoms can appear to be in one module as a result of problems somewhere else. In the best case, an error message will clearly identify the problem; otherwise, the problem will be assigned to the most likely area and perhaps transferred later. In any event, making the assignment correctly requires a fair amount of work and experience for the assigner, as is evidenced by the multiple layers involved in making the assignment. A second problem is load balancing: one engineer might have many problems to work on, while others have none.

An alternative basis for choosing engineers was found in a new support group that was set up during our study. Support engineers were not specialized by module, but were instead organized around change ownership, that is, an engineer assigned a particular problem report makes changes to any affected modules. As a result, task assignment can be done based on workload rather than specialization. In this case, a manager can make the assignment by tracking the status of individual engineers, or engineers can assign work to themselves whenever they finish a task. Many processes could be similarly redesigned to use generalists rather than specialists. For example, in a customer service process, the person who answers the phone could be enabled to resolve any problem rather than having to refer the problem to a specialist.

With change ownership, multiple engineers may have to work on the same module, thus creating a new shared resource dependency. This problem illustrates an important point: coordination mechanisms are themselves activities, and using a different kind of coordination mechanism to manage one dependency may create new dependencies that must in turn be managed. In this case, to manage these new task dependencies, the company implemented a source control system. The system maintains a copy of all source files. When engineers want to modify a file, they check it out of the system, preventing other programmers from modifying it. When the modification has been completed, the module is checked back in and the system records the changes made. The activities and analysis of this form are shown in Table 3.

The reorganization discussed above provided the substitution of a specialist mechanism for task assignment with a generalist mechanism. A more extreme substitution is to use a market-like task assignment mechanism. In this form, each problem report is sent to all available engineers. Each evaluates the report and, if interested in fixing the bug, submits a bid, saying how long it would take to fix the bug, how much it would cost or even what they would charge to do it. The task is then assigned to the lowest bidder, thus using information supplied by the engineers themselves as the basis for picking which engineer should work on the task. Many companies have out-sourced or switched to subcontractors for specific tasks, including some as central as engineering, customer service, or production,

Table 3 Activities in the Generalist Form of Task Assignment

Agent	Activity	Dependency Managed between...
Customer	Use system, find a bug	
	Report bug to response centre	Problem fixing task and capable actor
Response Centre	Look up bug in database of known bugs; if found, return fix to customer and stop	Problem fixing task and duplicate tasks
	Determine affected product and forward bug report to marketing engineer	Problem fixing task and capable actor
Marketing Engineer	Look up bug in database of known bugs; if found, return fix to customer and stop	Problem fixing task and duplicate tasks
	Attempt to reproduce the bug—part of diagnosing it determine affected module; if can't diagnose, forward to SWAT Team; if other product, forward to appropriate product manager; put bug report in the queue of bugs to work on	Problem fixing task and capable actor
Software Engineer	Start work on the next bug in the queue	Problem fixing task and actor's time
	Diagnose the bug	
	If it's actually an enhancement request, then treat it differently	
	Design a fix for the bug	
	If the change requires changes to a controlled document, then send the proposed change to the various managers and the change review board for their approval	Management of usability task and capable actor
Managers	Approve the change	Usability of fix by subsequent activities
Software Engineer	Check out the necessary modules; if someone else is working on them, then wait or negotiate to work concurrently	Problem fixing task and other tasks using the same module
	Write the code for the fix	
	Test the proposed fix	Usability of fix by subsequent activities
	Send the changed modules to the integration manager; check in the module	
Integration	Check that the change has been approved	Usability of fix by subsequent activities
	Recompile the module and link it with the rest of the system	Integration
	Test the entire system	Usability of entire system by next activity
	Release the new software	

although usually for larger units of work. Many industries rely almost entirely on subcontractors for individual tasks (e.g., construction, Italian textile manufacturing, and publishing).

To summarize, new processes for fixing bugs can be generated by substituting alternative coordination mechanisms for the ones used in a process. Some of these substitutions may be applicable to many processes, such as the alternative ways to manage approvals or task assignments, while others can be quite specific to the details of a particular process, such as a code management system. The process is recursive:

substituting one coordination mechanism for another may create new dependencies which in turn require additional coordination mechanisms.

4.3. Evaluating Alternative Organizational Processes

Before implementing any changes, it is important to evaluate the advantages and disadvantages of each kind of form. Clearly, this is hard to do in any general way, because the performance of a process depends heavily on its details and organization being studied. However, for the specific process changes discussed above, we can suggest one evaluation.

This section evaluates the three forms of task assignment considered above—specialist, generalist, and market-like—following Malone and Smith's (1988) analysis of four different organizational structures. In this model, tasks arrive at an organization and must be assigned to an actor who can execute them. (This model is general enough to represent many processes besides software bug fixing.) Malone and Smith compare organizational forms on three criteria: production cost (the average delay to process a task), coordination cost (the number of messages necessary to assign a task), and vulnerability of the form to failures of an actor (i.e., whether the organization still functions if one actor does not perform assigned tasks).

Many other factors could be added to complicate such a model. I will briefly consider three additional factors previously discussed: learning by engineers who work repeatedly on the same modules might reduce production costs; diagnosing a problem to choose an appropriate specialist and decomposing and distributing complex problems across specialists might increase coordination costs. Calculating these costs requires some detailed assumptions about parameters of the system, e.g., what proportion of tasks is complex or how long it takes to diagnose a problem versus sending a message. However, even without these assumptions, some qualitative comparisons can be made.

The first form, assignment based on specialists, has a low coordination cost. Assigning a task requires only four messages, from the customer to the service centre, from the service centre to the marketing engineer, from the marketing engineer to the engineering manager and from the manager to the software engineer. Each of these actors must evaluate the task and identify the appropriate specialist to work on it next.

Because software engineers are specialists, presumably they can quickly fix problems once assigned a task. However, problems that span modules must be decomposed and assigned to multiple engineers. If the load is distributed unevenly (i.e., some modules have more problems than others), then a problem may have to wait until the responsible engineer is free, increasing the time to finish the task. The engineer does not have to wait for the code to become available, however.

Finally, the form is vulnerable to the failure or overloading of a single actor because the engineer responsible for each module has no backup (in practice, of course, other engineers could try to fill in, although with greatly reduced productivity). Assignment based on module reinforces specializations by module, because engineers have little opportunity or need to learn about other parts of the system, and a large incentive to become expert in their modules.

The cost of the task assignment in the generalist model is also low, requiring the same number of messages. Furthermore, the final assignment is done by workload, eliminating the need for the marketing engineer to identify the specific module involved. Problems are handled by the next available actor, minimizing waiting time and reducing vulnerability of the organization to the failure of a single engineer. However, because the engineers are generalists, the time they take to fix a module is likely to be higher than in the specialist model. Because the organization does not take advantage of performance differences between actors, engineers do not have much incentive (or opportunity) to improve performance by learning about particular modules. Finally, if someone else is already working on a problem in the module, then the engineer will have to wait for the code to be available to make the changes.

The market-like model has a much higher coordination cost, because it requires many messages to assign each task (one for each bid request and bid). The cost of processing these messages includes, for example, the cost of having each engineer read each problem report. However, problems can be immediately assigned, although the engineer may have to wait for the code to be available to make the changes. Finally, in this model, the task will be assigned to the actor with the lowest bid, thus taking advantage of differences in knowledge. If the actors learn about a particular module, they can specialize, preferentially bidding for one type of task and constantly improving their performance on it. For example, an engineer who has recently worked on one module may be able to bid lower for other changes in that module.

The relative costs of these three forms are summarized in Table 4. Of course, researchers have identified additional factors that affect the feasibility of these forms. For example, the market-like form is susceptible to agency problems: if engineers are rewarded based

Table 4 Relative Costs of Different Task Assignment Mechanisms

Cost	Specialists	Generalists	Market-like
Production costs			
Waiting for engineer	Necessary	Unnecessary	Unnecessary
Waiting for module	Unnecessary	Necessary	Necessary
Fixing problem	Low	High	Low
Takes advantage of learning	On assigned modules	No	Yes
Coordination costs			
# of diagnoses	4	2	$2 + N$
# of messages to assign	4	3	$2N$
Decomposition and assignment of subtasks	Necessary	Unnecessary	Unnecessary
Vulnerability to failure	High	Low	Low

Note: N is the number of software engineers.

on the number of bugs they fix, they might bid unrealistically low to win assignments; if they are paid a flat salary, they might not bid at all. As with the product of any redesign method, the implications of such factors must be considered before a particular form can be recommended.

4.4 Effects of Electronic Media on the Choice of Coordination Methods

Coordination theory provides a useful theoretical framework for analyzing the implications of new communications technologies. An organization's choice of coordination mechanisms is affected by its relative costs, which in turn depend on the technology available. The use of electronic media (and other kinds of IT) changes the relative cost of coordination mechanisms, making new processes feasible or desirable. Coordination theory thus provides a conceptual link between organizational form and the use of communication technology.

- For *task assignment*, communications technology makes it easier to gather information about available resources and to decide which resources to use for a particular task. At a macro level, Malone, Yates, and Benjamin (1987) suggest that decreased coordination costs favour more extensive use of markets, which usually have lower costs but require more coordination activities, over vertical integration, which makes the opposite trade-off.
- Avoiding *duplicate tasks* is difficult if there are numerous workers who could be working on the same task. For example, in a software company, the same bug may be reported by many users; the company would prefer not to diagnose and solve this problem repeatedly. Past solutions to this problem include centralizing the workers to make exchange of information easier, specializing workers so that identical tasks are all assigned to the same worker or simply accepting the duplication. New alternatives include an information system containing information about tasks and known solutions or communications technologies, such as a computer conferencing system, that can cheaply broadcast questions to a large community (Finholt and Sproull 1990).
- Just-in-time delivery of components, a new way to manage *prerequisite dependencies* between suppliers and users, is in large part a communications innovation: new information transmission methods replace keeping inventories on hand in the plant.
- For *sharing information resources*, communications and database technologies may automate the necessary coordination mechanisms. For example, coordination is necessary if multiple tasks use common information stored on paper (a shared resource dependency). It may therefore be desirable to have a single individual handle all the data, to simplify the coordination. For example, a conference room schedule is usually kept in a central location because of the possibility of conflicting reservations and the prohibitive cost of updating copies every time a reservation is made. Data such as customer accounts or credit information are often handled similarly, resulting in specialization of actors based on their access to information. Database and communications systems enable multiple workers to access and make changes to data. By empowering workers and reducing the need for specialization, IT can change the basis for assigning tasks. For example, if all workers are equally capable of performing a task, then tasks can be assigned on criteria such as workload or the customer involved, rather than on availability of data or specialization. Such a change was made to the Citibank letter of credit process when a group of specialists, each performing a single step of the process, were replaced by generalists who handle the entire process for particular customers (Matteis 1979).

Clearly, the exact form of the technology is less important than the functionality it provides. In the case discussed in this paper, for example, the main communications channel between groups is actually a database of change requests rather than a more conventional system like electronic mail or computer conferencing. (Electronic mail was available, but not heavily used within the software development group.) However, the database system provides much of the same functionality as electronic mail and was sometimes even used in the same way. For example, on one occasion the response centre created a new change request to enter a question about the status of an older report; the responsible engineer then replied to the question in another field of the database and closed the request.

Thus, rather than focusing on specific technology, one approach to analyzing such technologies is to consider which of a system's attributes are important. Nass and Mason (1990) discuss numerous dimensions of communications technology; key attributes for the case above include permanence across time, one-to-one vs. one-to-many communication and programmability and integration with computer technology.

Permanence across time means that messages entered into the system can easily be retrieved later. Computer conferencing and databases have this property; telephones and ordinary electronic mail do not (although messages from both can be archived). This function allows the product of fixing a problem to be stored and reused, a key part of one of the coordination mechanisms discussed above.

A second key characteristic is the number of possible recipients for a single message. Telephones are usually one-to-one; paper memos can be one-to-many, for a cost; and electronic media can be one-to-many with almost no extra cost. This functionality enables more coordination intensive forms. For example, in the market-like form, the response centre needs to send the same message (a bid request) to all software engineers. The organization could use a computer bulletin board on which task announcements are posted to support this communication. Such a system would reduce the coordination cost by replacing multiple bid request messages with a single broadcast.

Finally, electronic communications media may be programmable or integrated with computer technology, potentially automating parts of certain coordination mechanisms. For example, such a system could filter problem reports for engineers based on an interest profile, reducing the number that need to be evaluated in a market-like task assignment mechanism. Bid processing and awarding could also be easily automated, further reducing the cost of a market-like mechanism, perhaps enough to make it desirable.

5. Conclusion

Engineering change provides a microcosm of coordination problems and mechanisms to solve them. Successful implementation of a change requires management of numerous dependencies among tasks and resources. A variety of mechanisms are used to manage these dependencies. For example, the possibility of duplicate tasks may be ignored or may be investigated before engineers attempt to solve the problem. Dependencies between tasks and the resources needed to perform them are managed by a variety of task assignment mechanisms, such as managerial decision-making based on expertise or workload; those between modules of the system, by technological coordination mechanisms, such as source control systems.

The choice of coordination mechanisms to manage these dependencies results in a variety of possible organizational forms, some already known (such as change ownership) and some novel (such as bidding to assign problem reports). The relative desirability of mechanisms is likely to be affected by the use of electronic media. For example, the use of a computer system may make it easier to find existing solutions to a problem, either in a database or from geographically distributed coworkers. Such a system could reduce both duplicate effort and coordination costs.

The software change process has interesting parallels in other industries. Despite differences in the products, other engineering change processes studied by this author (Crowston 1991) had similar goals, activities, coordination problems, and mechanisms. Further afield, one reviewer noted parallels between diagnosing software bugs to assign them to engineers and diagnosing patients to assign them to medical specialists. An analysis similar to the one presented here might reveal interesting alternatives in the medical diagnosis domain as well. Such an effort may be particularly timely, given the leading role IT-enabled changes play in some proposals to revamp health care systems.

Coordination theory, like all theories, is a simplification of the complexity of real organizations. It describes a variety of alternative processes, while highlighting the contribution of new communications media and other information technologies. The single example presented here demonstrates the potential of this approach. However, the suggestions of the analysis need to be tempered by consideration of omitted factors. The technique focuses on how tasks are performed, rather than how employees are motivated to perform, come to understand their jobs, or develop shared cultures. For example, a lower mechanism cost does not mean that that mechanism is always better or should be implemented. As mentioned above, market-like task assignment mechanisms have certain cost benefits, but are also susceptible to agency problems that must be addressed if they are to succeed. Rather than saying what must happen, the analysis suggests possibilities that an informed manager can consider and modify to fit the particulars of the organization.

Said alternatively, coordination theory does not make strong predictions about what must happen to any single organization that implements a new communication system, although it does suggest what will happen in aggregate (Malone et al. 1987). Rather than the specific accuracy of its predictions, therefore, an appropriate test for the theory is its utility for organization designers. Coordination theory is a success if those attempting to understand or redesign a process find it useful to consider how various dependencies are managed and the implications of alternative mechanisms. As an example, we are currently using these techniques to compile a handbook of organizational processes at a variety of levels and in different domains (Malone et al. 1993). Managers or consultants interested in redesigning a process could consult the handbook to identify likely alternatives and to investigate the advantages or disadvantages of each. Coordination theory makes the handbook feasible by providing a framework for describing more precisely how processes are similar and where they differ.

A redesign agenda suggests several additional research projects. First, development of the handbook and general use of a coordination-theory analysis require more rigorous methods for recording processes and identifying dependencies in organizations. There are already many techniques for data collection that are relevant, but none focus explicitly on identifying dependencies. Other researchers affiliated with the handbook project have proposed an approach that relies on basic techniques of ethnographic interviewing and observation to collect data and activity lists to identify dependencies and coordination mechanisms (Pentland et al. 1994). Prototypes of such methods are currently being used for our research and in the classroom. Experiences to date attempting to teach students to use this technique indicate that it takes a while to pick up the ideas, but that using them leads to greater insight into the process.

Second, more work is needed to elaborate the typology of dependencies, particularly those between objects, and associated mechanisms. Identifying additional mechanisms is an inevitable result of the work being done to record a variety of processes, and I expect that better ways to organize these mechanisms will be developed. Finally, computer simulations of processes will provide an aid to understanding the performance of processes using alternative coordination mechanisms and might even automate the exploration of alternative forms.

Although still under development, coordination theory seems to provide a much-needed underpinning for the study and design of new organizational processes. The result of these efforts will be a coordination-theory based set of tools for organizational analysts and designers, which perhaps will help realize the potential of electronic media and new organizational forms.

Acknowledgment
This research was supported by the Center for Coordination Science at the Massachusetts Institute of Technology and a fellowship from the Ameritech Foundation through the University of Michigan Institute for Public Policy Studies. The field work and preliminary analysis described in §3 was done with the assistance of Stephen Brobst. The paper has benefited from discussions with Michael Cohen, Jin-tae Lee, Thomas Malone, Charlie Osborn, Brian Pentland, and other members of the CCS Process Handbook Project, comments from Gerardine DeSanctis and anonymous reviewers, editing by Nancy Kotzian, and from repeated and careful reading by Marie Williams.

Appendix A

The Software Change Process

The software maintenance process starts when a problem is found. If a customer calls the customer support centre with a problem, the call handler tries to solve it using manuals, product descriptions, and the database of known problem. If the problem appears to be a bug that is not already in the database, a new problem report is entered. Many problems are found during the development process by the testing group, who enter problem reports directly.

Marketing Engineer. A marketing engineer for the affected product reviews the problem report for completeness and attempts to replicate the problem. The marketing engineer may decide that the problem is really a request for an enhancement, which is handled by a separate process. If the bug is genuine, the marketing engineer determines the location of the problem and assigns the problem report to the software development unit responsible for that module.

Software Engineer. A coordinator in the development unit assigns the problem report to the appropriate software engineer, who investigates the problem. If the problem turns out to be entirely in another module, then the engineer passes the request to the engineer responsible for the other module. If the problem is internal to a single module, the engineer just fixes the module. If the problem requires changes to multiple modules, the engineer discusses the changes with the owners of the affected modules (as well as other interested engineers) and arranges for them to modify their modules. All changes require the approval of management. Changes to interfaces intended for general use require a design review and approval from a change review board.

Integration and Testing. When the engineer is satisfied with the change, he or she submits the new code to the testing and integration group. The integration group then recompiles the changed code and relinks the system. The kernel is then tested; any bugs found are reported to the engineer, potentially starting another pass through the process. Customers are periodically sent the most recent release of the system. In some cases, they receive a patch for a single change.

References

Abbott, A. (1992), "From Causes to Events: Notes on Narrative Positivism," *Sociological Methods and Research*, 20, 4, 428–455.

Abell, P. (1987), *The Syntax of Social Life: The Theory and Method of Comparative Narratives*, New York: Clarendon Press.

Crowston, K. (1991), *Towards a Coordination Cookbook: Recipes for Multi-Agent Action*, unpublished doctoral dissertation, Cambridge, MA: MIT Sloan School of Management.

Davenport, T. H. and J. E. Short (1990), "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review*, 31, 4, 11–27.

Dennett, D. C. (1987), *The Intentional Stance*, Cambridge, MA: MIT Press.

Finholt, T. and L. S. Sproull (1990), "Electronic Groups at Work," *Organization Science*, 1, 1, 41–64.

Galbraith, J. R. (1977), *Organization Design*, Reading, MA: Addison-Wesley.

Hammer, M. (1990), "Reengineering Work: Don't Automate, Obliterate," *Harvard Business Review*, 68 (July–August), 104–112.

____ and J. Champy (1993), *Reengineering the Corporation: A Manifesto for Business Revolution*, New York: Harper Business.

Harrington, H. J. (1991), *Business Process Improvement: The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*, New York: McGraw-Hill.

Harrison, D. B. and M. D. Pratt (1993), "A Methodology for Reengineering Business," *Planning Review*, 21, 2, 6–11.

Kidder, L. H. (1981), *Research Methods in Social Relations* (4th ed.), New York: Holt, Rinehart and Winston.

Lawler, E. E., III (1989), "Substitutes for Hierarchy," *Organizational Dynamics*, 163, 3, 39–45.

Lientz, B. P. and E. B. Swanson (1980), *Software Maintenance Management: A Study of the Maintenance of Computer Applications Software in 487 Data Processing Organizations*, Reading, MA: Addison-Wesley.

Malone, T. W. and K. Crowston (1994), "The Interdisciplinary Study of Coordination," *Computing Surveys*, 26, 1, 87–119.

____, ____, J. Lee, and B. Pentland (1993), "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes, in *Proceedings of Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Morgantown, WV: IEEE Computer Society Press, 72–82.

____ and S. A. Smith (1988), "Modeling the Performance of Organizational Structures," *Operations Research*, 36, 3, 421–436.

____, J. Yates, and R. I. Benjamin (1987), "Electronic Markets and Electronic Hierarchies," *Communications of the ACM*, 30, 484–497.

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

Matteis, R. J. (1979), "The New Back Office Focuses on Customer Service," *Harvard Business Review*, 57, 146–159.

McKelvey, B. (1982), *Organizational Systematics: Taxonomy, Evolution, Classification*, Berkeley, CA: University of California.

____ and H. Aldrich (1983), "Populations, Natural Selection and Applied Organization Science," *Administrative Science Quarterly*, 28, 101–128.

Mohr, L. B. (1982), *Explaining Organizational Behavior: The Limits and Possibilities of Theory and Research*, San Francisco, CA: Jossey-Bass.

Nass, C. and L. Mason (1990), "On the Study of Technology and Task: A Variable-based Approach," in J. Fulk and C. Steinfield (Eds.), *Organizations and Communication Technology*, Newbury Park, CA: Sage, 46–67.

Osborn, C. (1993), *Field Data Collection for the Process Handbook*, unpublished working paper, Cambridge, MA: MIT Center for Coordination Science.

Pentland, B. T. (1992), "Organizing Moves in Software Support Hotlines," *Administrative Science Quarterly*, 37, 527–548.

____, C. S. Osborn, G. M. Wyner, and F. L. Luconi (1994), *Useful Descriptions of Organizational Processes: Collecting Data for the Process Handbook* (working paper 190), available from Center for Coordination Science, E40-170, MIT, Cambridge, MA 02139.

Powell, W. W. (1990), "Neither Market nor Hierarchy: Network Forms of Organization," *Research in Organizational Behavior*, 12, 295–336.

Rich, P. (1992), "The Organizational Taxonomy: Definition and Design," *Academy of Management Review*, 17, 4, 758–781.

Sanchez, J. C. (1993), "The Long and Thorny Way to an Organizational Taxonomy," *Organization Studies*, 14, 1, 73–92.

Tushman, M. and D. Nadler (1978), "Information Processing as an Integrating Concept in Organization Design," *Academy of Management Review*, 3, 613–624.

Accepted by Gerardine DeSanctis; received September 1993. *This paper has been with the author for three revisions.*