Electronic communication and new organizational forms: A coordination theory approach

by

Kevin Crowston

CCS TR #175, Sloan WP # 3719-94

August 1994
Electronic communication and new organizational forms:
A coordination theory approach

Kevin Crowston

School of Business Administration
University of Michigan

August 17, 1994

Author's present address:
School of Business Administration
University of Michigan
701 Tappan
Ann Arbor, MI 48109-1234 USA
crowston@umich.edu
+1 (313) 763-2373
+1 (313) 763-5688
Electronic communication and new organizational forms:  
A coordination theory approach

Abstract

Describing and categorizing organizational forms remains a central problem in organization theory. Unfortunately defining organizational form poses numerous difficulties. Rather than attempting to categorize entire organizations, researchers have instead suggested focusing on how particular tasks are performed, i.e., adopting the process as the unit of analysis. An important practical problem then is to identify processes that would be suitable for performing a desired task, especially processes that are enabled by the use of new electronic media and other forms of information technology.

Coordination theory provides an approach to the study of processes. In this view, the form a process takes depends on the coordination mechanisms chosen to manage dependences among tasks and resources involved in the process. These mechanisms are primarily information-processing and so the use of new media will particularly affect their cost, perhaps changing which are preferred.

In this paper, I use coordination theory to analyze the software change process of a large mini-computer manufacturer and suggest alternative ways the dependences involved could be managed and thus alternative forms the process could take. Mechanisms analyzed include those for task assignment, resource sharing and managing dependences between modules of code. The organization studied assigned problem reports to engineers based on the module which appeared to be in error; engineers specialized in particular modules. The framework suggests alternative mechanisms including assignment to generalists based on workload or based on market-like bids. Modules of code were not shared, but rather "owned" by one engineer, thus reducing the need for coordination; a more elaborate code management system would be
required if multiple engineers needed to work on the same modules. Finally, engineers managed
dependences between modules informally, based 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 the coordination theory for exploring the space of organizational forms. 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 perhaps automatically suggesting
potential alternatives.
1 Introduction

Describing and categorizing organizational forms remains a central problem in organization theory (e.g., McKelvey, 1982; Rich, 1992; Sanchez, 1993). Unfortunately, defining organizational form poses numerous difficulties. Rich (1992) criticizes typologies based on single characteristics, such as production technology (Perrow, 1967; Woodward, 1980), as being too simplistic. He suggests instead viewing organizational form as a multi-dimensional construct and grouping together organizations that share similar sets of characteristics. As McKelvey and Aldrich (1983) point out, however, most large organizations are actually mixtures of different forms. Mohr (1982) describes organizational structure as, "multidimensional—too inclusive to have constant meaning and therefore to serve as a good theoretical construct" (p. 16).

In short, an entire organization is too aggregate a level of analysis for meaningful comparison. Instead, researchers have suggested focusing on how particular tasks are performed, i.e., adopting the process as the unit of analysis (Mohr, 1982; Abbott, 1992, p. 428–9). For example, rather than listing ways in which General Motors and Ford are alike or different, researchers might compare their processes for designing automobiles or even more specific subprocesses. The problem thus becomes, not what form does an organization have, but what process does it use to accomplish particular tasks?²

Given a task being performed by an organization, an important practical problem is to identify other processes that would also be suitable for performing that task. As companies scramble to adapt to increasingly frequent environmental changes, this question has become even more pressing. For example, a manager may realize that the survival of his or her company depends on reducing time-to-market and improving quality, but still find it difficult to translate these goals into concrete organizational changes (e.g., as part of a business process redesign effort

² Such an analysis presumes, of course, that organizations perform tasks. This presumption can be questioned, but it seems clear that many organizations have at least a stated purpose that would be preserved in a reorganization.
Other managers may be concerned with making effective use of the billions of dollars invested in different kinds of information technology (IT), electronic media in particular, and wonder what kinds of organizational forms become possible as the historic constraints on communications and information processing are relaxed. Underlying all of 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) reporting process: customers having problems with a piece of software report the problems to its developers, who (they hope) will eventually provide some kind of solution. One company I studied, the developer of a mini-computer operating system, has an elaborate process to receive problem reports, filter out reports 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 to avoid the problem; the final software engineer might develop a patch (i.e., a change to 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 below.)

Why is the process structured this way, with finely divided responsibility for different parts of the process? What different forms are there for this process? If we examine many companies, we will observe a wide variety of approaches to this same software bug fixing process. For example, other companies (and even other parts of the company I studied) use what is sometimes called change ownership (Swanson and Beath, 1990): when a problem report arrives, it is simply assigned to the next free engineer. Any engineer can, in principle, fix any module and task assignments are therefore based on workload rather than specialization. If we examine many processes, we will see an even wider range of possible forms. For example, individuals or firms may be generalists, performing a wide variety of tasks, or specialists,
performing only a few. Tasks may be assigned to actors within a single organization, as with bug fixing; others take place in the market, as with auditing, consulting and an increasingly wide variety of services; and increasingly corporations are forming networks for allocating work (e.g., Powell, 1990).

When we begin to systematically compare these processes, however, patterns emerge. If the organizations are performing essentially the same task, typically the same basic steps are required. For example, all software companies that respond to bug reports need to diagnosis the bug, write code for a fix, test the fix and integrate it with the rest of the system. Looking more broadly, many engineering change processes have similar steps. While the general steps are the same, organizations differ in important details of how these large abstract tasks are decomposed, who performs them and how they are assigned, that is, in how these tasks are coordinated. Even here there are common patterns: similar problems arise and are managed by similar coordination mechanisms.

2 A coordination theory approach to organizational form

To study coordination, I use the model presented by Malone and Crowston (1994), who define coordination as “managing dependences between activities” (p. 90) and 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, the actors are the customers and various employees of the software company; activities include those mentioned above. The goal of the process in this case appears to be eliminating problems in the system, but

---

3 In some cases, a group of individuals may be represented by a single actor (Abell, 1987, p. 13); for example, to simplify the analysis of a particular subunit, the other subunits with which it interacts might all be represented as collective actors.
alternative goals—such as appearing responsive to customer requests—could also be analyzed. Finally, resources include the problem reports, information about known problems, computer time, software patches, source code and so on.

In this view, actors in organizations face coordination problems arising from dependences that constrain how tasks can be performed. These dependences 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) 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 work in the form of coordination mechanisms. For example, a software engineer planning to change one module in a computer system must check that the changes will not affect other modules or arrange for any necessary changes to modules that will be affected; two engineers working on the same module must each be careful not to overwrite the other’s changes. Coordination mechanisms may be quite specific, such as a code management system to control changes to software, or quite general, such as hierarchical or market mechanisms to manage assignment of activities to actors or other resources. Note that many coordination mechanisms are primarily information processing activities (e.g., ordering or picking tasks, negotiating with other actors or informing them about planned activities) and thus potential candidates for support from electronic media.

---

4 In taking this approach, we adopt Dennet’s (1987) intentional stance: since 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. For instance, in analyzing markets, we might as observers regard the goal to be achieved as one of optimally allocating resources to maximize consumer utilities (e.g., Debreu, 1959). Even though no single individual in the market necessarily has this goal, observers might evaluate market coordination in terms of how well it is achieved.

Draft of 8/17/94
In general, there may be several coordination mechanisms that could be used to address a given dependence; different organizations may use different mechanisms to address similar problems, thus resulting in a different organizational form. Given a particular organization performing some task then, one way to generate alternative processes is to identify the particular dependences and coordination problems faced by the organization and consider what alternative coordination mechanisms could be used to manage them. For example, Lawler (1989) argues that the functions of an organization's hierarchy, many of which are ways of coordinating the actions of the lower levels, could be accomplished in other ways, such as work design, information systems or new patterns of information distribution. (Of course, there are also many non-coordinating functions of the hierarchy, such as motivating or coaching workers that must be considered before making such changes.)

2.1 Implications of coordination theory for the study of electronic media and organizational form

Coordination theory provides a useful theoretical framework for analyzing the implications of new communications technologies. An organization's choice of coordination mechanisms (and thus in our view, the organizational form) is affected (although not determined) by their relative costs, which in turn depends on the technology available. The use of electronic media (and other kinds of IT) will change 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 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, instead of vertical integration, which make 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 is may be reported by many users; the company would prefer to not 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 that can cheaply broadcast questions to a large community, such as a computer conferencing system (Finholt and Sproull, 1990).

Just-in-time delivery of components, a new way to manage prerequisite dependences between suppliers and users, is in large part a communications innovation: new ways to transmit up-to-date information about what components should be delivered replace keeping inventories on hand in the plant.

For sharing information resources, communications and database technology may themselves provide the necessary coordination. For example, coordination is necessary if multiple tasks need access to information stored on paper (a shared resource dependence). 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 being made 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.

A database and communications system enable multiple workers to access and make changes to the data. By empowering workers and reducing the need for specialization, IT can change the basis for assigning tasks; 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

Draft of 8/17/94
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).

2.2 A typology of coordination mechanisms

As a guide to such analyses, Crowston (1991) presents a preliminary typology of dependences and associated mechanisms. The main dimension of the typology is the type of objects involved in the dependence, either tasks (goals and activities from Malone and Crowston's (1994) framework) or resources used or created by tasks (including the effort of the actors). There are therefore three major kinds of dependences between two objects: those between two tasks, between two resources or between a task and a resource. These categories can be further refined: for example, task-task dependences are distinguished by considering that kind of resource or resources are shared by the two tasks and how they are used (as an input or as an output of the task). Mechanisms also differ by purpose, either to manage constraints created by a dependence or to maintain a desired dependence.

The typology of dependences and examples of associated coordination mechanisms is shown in Table 1. Some of the cells of this typology are more developed than others; for example, task-task dependences have been analyzed in some detail, while resource-resource dependences have not.

Insert Table 1 about here

3 Data and methods

As a concrete example of a coordination theory analysis, I will examine a software change process and consider alternative forms the process could take. The particular organization in this example is the minicomputer division of a large diversified electronics corporation; in 1989, when the study started, the entire corporation had sales of approximately $10 billion and roughly
100,000 employees\textsuperscript{5}. The computer division produces several lines of minicomputers and workstations and develops system software for these computers.

3.1 \textit{Research setting}

This process was examined as part of a study of the engineering change process in three organizations. Engineering change processes are interesting for several reasons. First, the process is very coordination-intensive and requires individuals in engineering and manufacturing to work together to be effective. Second, even though each change is unique, change management is relatively routine. Pentland (1992) notes advantages to studying this sort of routine work: each change forms a clearly bounded unit of work and intensive records are kept of each one. Finally, change management is done in some form by nearly all manufacturing companies, many of which are interested in improving the performance of this process.

The particular group studied 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. Development of the operating system had started about 5 years before we began our study; at that time, the system had just been initially released.

To set the context for my analysis, I will first briefly describe the product. The operating system is the basic software of the computer; its major function is to insulate programmers from the details of the hardware. Additional mechanisms 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. Operating systems are typically quite specific to the details of the hardware; the system studied works only with this manufacturer's hardware.

\textsuperscript{5} The field work and analysis was done with the assistance of Stephen Brobst.
Software is interestingly different from other products. One of the most fundamental differences is that software is not a physical product, which has several implications. First, once the development process is completed, reproduction of the finished product is relatively straightforward, consisting mostly of duplicating tapes and documentation. As well, the product is very malleable: almost any change that can be imagined can be made without the physical constraints of other products, such as the need to change tooling or produce new components. (This is not to imply that changes to software are costless, however.) As a result, the rate of changes is higher in software than in hardware (A, p. 110).6

Second, problems are much more likely to be systematic. 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. A problem with a piece of software, on the other hand, it is likely to occur in every copy of the software. This is especially true for system software, which is usually less customized than mainframe or mini-computer application software and thus less variable, and for micro or minicomputer software, where the variance in the underlying hardware is less. In a sense there is only one product, not multiple instances. As a result, software change processes are important to do well, because they affect every user.

The operating system developed by this organization is decomposed hierarchically into several major subsystems, such as the process manager or file system; each subsystem is further divided into a number of modules. Each module implements a small set of services. 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, (parts of) the process management code depends on (part of) the file system code. The set of routines and data provided by a module form the interface to that module.

6 References to page numbers of my field notes are given as the source for quotations or statements by subjects.
Different interfaces are provided for use by different classes of users. Interfaces provided to customers are described in published manuals for the system and rarely if ever change. Other interfaces, called service interfaces, are provided for general use by other parts of the system software. Service interfaces used by other development groups are documented in external specifications; those used only within a single unit may be documented in an internal specification or perhaps not at all. Manuals and external specifications are maintained in a paper documentation library. Programmers who request copies of a document are registered as a user of the described interface, in principle so they can be informed of any changes to the document. At the time of our study, there were 800-900 documents, with about 1000 users total. A total of 15,000 copies of documents had been distributed.

3.2 Data collection

The analysis presented here is based on 16 interviews with 12 different individuals, including 6 software engineers, two managers and three members of support groups and one marketing engineer. The interviews were carried out during 6 trips to the company's engineering headquarters; most were one to two hours long. As well, I worked on the analysis with a former member of the software development group who provided much additional information.

As discussed above, coordination mechanisms primarily involve information-processing activities. For my study I therefore adopted the information processing (IP) view of organizations (e.g., March and Simon, 1958; Galbraith, 1977; Tushman and Nadler, 1978) because of its focus on how organizations process information.

During the interviews, I attempted 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 or (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 being studied. This identification was done with the aid of a few key informants, and refined as the study progressed. As well, formal documentation of the process was used as a starting point where it was available. For example, a number of individuals design and code parts of the operating system, all working in roughly the same way and using the same kinds of information; each is an example of a "software engineer actor." Response centre or marketing engineers use different information and process it differently and were therefore analyzed separately.

Subjects were identified by the key informants, based on their job responsibilities; there was no evidence, however, that their reports were atypical. I then interviewed each subject to identify the types of information received by each kind of actor and the way each type is handled. Data were collected by asking subjects: (1) what kinds of information they receive; (2) from whom they receive it; (3) how they receive it (e.g., from telephone calls, memos or computer systems); (4) how they process the different kinds of information; and (5) to whom they send 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 created and exchanged as part of the process or that described standard procedures or individual jobs. Not surprisingly, the process as performed frequently differs from the formally documented process. For example, at a different site in the study, engineers receive a listing of all approved changes, but the official list seems merely to confirm that the changes have been approved. In order to react to a change an engineer must be warned of it well in advance of its appearance on the official list. This warning seems to happen primarily through an informal process. It is this informal process 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 describes the actors involved, which steps each performs and the information they exchange. 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 other groups in the past or who interacted with them is that processes are similar in other software development units, but I have no direct information about them.

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 really not know the answer or may be mistaken.

Some of these biases can be controlled by cross-checking reported data with other informants. For example, if one interviewee reports sending information to a particular group, I can check if that other group reports receiving such information.

Furthermore, the modelling process serves as another check on the consistency of the data. I used an iterative approach, sometimes called the negative case study method [Kidder, 1981 #114], switching between data collection and model development. The initial round of data collection serves as the basis for an initial model. Constructing this model revealed omissions in the data, for example, places where it was not clear how an actor reacts to some message or from whom a particular piece of information comes. These omissions or ambiguities serve as the basis for further data collection.
3.4 The change process

Reasons for changes. 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 this company, problems are defined as disagreements between the behaviour of a program and its documentation. The interface and behaviour of a module are supposed to match the document describing it, so changing the interface requires a revision of the corresponding document. Fixing problems usually requires changing the software, but sometimes the fix is made to the documentation instead. Perfective changes are those made to improve a correct program without altering its behaviour, typically by improving performance. Adaptive changes are those that add new functionality, altering the software to meet changing requirements. The last two categories of changes are mostly made to enhance the system rather than as part of the change process, so I will not discuss them further in this paper.

Goals of the change process. The organization had the following stated goals for the change process (A4, p. 7):

- 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; 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

More generally, I believe the change process actually has two primary goals: to 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" (A, p. 108).
Many problems are found during the development process by the testing group. Some are found by customers using the product. The software maintenance processes start when the testing group or a customer notices some problem with a product and complains. The testing group can enter a problem report directly in the change request system described below. Customer complaints are entered into the system by a field engineer or by the customer support centre staff if the customer telephones for help.

When a customer calls, the call is directed to one of approximately 300 specialists in one of 10-15 groups, depending on the reported product. The staff member handling the call may use information such as the manuals distributed with the product, marketing information, current price lists, ordering information and other documents describing the product. Specialists also have access to a database of all calls logged in the past few years and a database of change requests.

Problems may be due to several causes, such as user misunderstanding of the product (an estimated 4 out of 6 calls per day, according to one interviewee), hardware problems (1/2 to 1 call per day) or real problems with the software (1 to 1 1/2 calls per day). The reported problem may duplicate a known report already in the change request database. In this case, if there is a patch, the call handler can order it for the customer. If the reported problem does not match any existing problem report, the call handler can enter a new change request in the change request tracking system. Our interviewee had entered a change request 2 times in 9 months of working in a specialist group (i.e., on the order of 2 out of an estimated 1000 calls). The change request database is the major communication channel between the customer service centre and the development engineers.

Marketing engineer. Once a change request is entered from any source, it is retrieved by a marketing engineer for review. There are eight such engineers for the operating system we studied. The marketing engineers get a list of problems entered every morning. About half the
time the problem report is complete enough to work with; in other cases, the engineer requests additional information from the submitter and waits for it to be sent.

Once the report is complete, the marketing engineer attempts to replicate the problem to gain a better understanding of its causes. The engineer may also determine that the problem is really a user misunderstanding, a duplicate of a known problem or due to a documentation error. If the problem appears to be genuine, it gets processed further. The marketing engineer may also decide that the problem is really a request for an enhancement, which are handled by a separate process. The marketing engineer then sets the priority for fixing the problem.

The marketing engineer next determines the location of the problem and assigns the change request to the development unit responsible for that module. If the marketing engineer can not locate the problem, the report goes to a group called the SWAT team. This team has access to the system source code which they use to further diagnose the problem. At the least, they locate the bug within some module and assign the report to the appropriate engineering group; they may go as far as to suggest a possible fix to the engineer, although they do not make changes themselves.

Software engineer. Periodically a coordinator in each software development group assigns new change requests in the database to the engineer responsible for the affected modules. The software engineer then 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. The engineers first discuss the problem informally; if it is agreed that the problem should be transferred, the engineer then changes the assignment in the change request tracking database.

If the problem is internal to a single module, the engineer can just fix the module and resubmit the code to the integration group. In other cases, the change may require changes to an interface used by other modules, requiring that changes be made to those modules as well. In
these cases, the engineer must discuss the change with the owners of the affected modules as well as other interested engineers and arrange for them to change their modules as necessary.

*Change approval process.* Changes to some service interfaces (those intended for general use) require changes to documents that are controlled by a change control group. To change the interface and the related document, the engineer writes a proposal for the change. For a major change, this might be a thick document which is widely circulated; for a minor change, only a few people may be involved. Typically the change includes a marked up copy of the document indicating what will be changed. The engineer investigates who will be affected by the change and invites these people to the design reviews. If they are not affected, they simply say so; otherwise, they come to the design review and comment on the proposed change. When the change is agreed on, the engineer seeks approval from management and from the change review board. The board is composed of representatives of different software development units, software manufacturing, integration and a testing and performance unit, 15-20 people in total. This group meets once a week for two hours, during which they discuss change requests, among other issues. For this operating system, the board considered 4-5 changes per week to documented interfaces.

Once the change is approved, the engineer changes the code as necessary. The end result is code for a new module with the problem fixed. When the change is complete, the engineer and another engineer review the code to check that no other errors were introduced. As well, the engineer tests the affected modules by recompiling it and relinking the system (i.e., merging the code for all modules to create the complete system). For changes that affect multiple modules, the first engineer informally tests the entire change with the aid of the other engineers.

*Integration and testing.* When the engineer is satisfied with the change, he or she submits the new code to the testing and integration group. In order to submit a change, the engineer must identify which change request is being addressed; without a request, the integration group will not accept the change. The submittal form must also be approved by the engineer’s manager.
When multiple modules must be changed, the changes for each module are submitted separately, but with a notation that it is part of a bundle. The initial engineer indicates when the bundle is complete. The integration group then recompiles all 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.

If the problem was serious and required a quick response, a fix can be released in the form of a patch. To make a patch, the compiled code for the just the affected modules is copied on a tape which can be separately installed on the affected customer's system by a field engineer or the customer. If the problem did not need an immediate solution (e.g., there is a workaround that avoids the problem), then the customer would wait for a routine release of the system that includes the change. Customers are periodically sent the most recent release of the system.

The activities performed for a typical change are summarized in the first two columns of Table 2. Although no particular bug is necessarily treated in exactly this way, these activities were described as typical by my interviewees.

*Insert Table 2 about here*

4 Dependences and coordination mechanisms in software bug fixing

In the course of making a change to the system, numerous dependences must be managed. The coordination theory approach to analyzing and redesigning this process suggests identifying these dependences and considering alternative ways to manage them. There are (at least) two ways to identify dependences and coordination mechanisms (Osborn, 1993).

First, we can examine activities in the current process, identify those that are seem to be part of some coordination mechanism and determine what dependence they manage. Some of the

---

7 Because the original study focused on the actions of the software engineers, the response centre was treated as a collective actor. As a result, activities internal to it, such as the task assignment discussed above, are omitted from the summary of the process.
activities in the bug fixing process appear to be instances of the coordination mechanisms discussed earlier; the dependences such activities apparently manage are listed in the third column of Table 2. For example, one of the first things the customer service centre staff, marketing engineers and software engineers do when receiving a problem report is check if it duplicates a known problem listed in the change database. In the typology, looking for duplicate tasks is listed as a coordination mechanisms for managing a dependence between two tasks which 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 is the case in this example).

Task assignment is a coordination mechanism for managing the dependence between a task and a performer by finding a performer to do 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. Prioritizing tasks, performed by the marketing and software engineer, is a sign of a resource allocation mechanism.

A second approach is to list the tasks and resources involved in the process and consider what dependences are possible between them. It may be that some of the steps in a process are coordination mechanisms for managing those dependences. 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. Resources include the problem reports, the efforts of a number of specialized actors and the code itself.

Dependences between tasks can be identified by looking for resources used by more than one task. For example, many tasks create some output, such a bug report, a diagnosis or new code, that is used as input by some other task, thus creating a prerequisite dependence between the two. Malone and Crowston (1994) note that such dependences often impose usability and
inventory constraints. Some of the 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.

If there are two problems in the same module, then both bug fixing tasks need the same code. In this process, this dependence is managed by assigning modules of code to individual programmers and then assigning problems in these modules to that programmer. This arrangement is often called "code ownership," since each module of the system has a single owner who performs all tasks that modify that module. Such an arrangement eliminates or at least greatly reduces the need for coordination to share that resource.

Finally, there are dependences between modules owned by different engineers that constrain what changes can be made and must therefore be managed. Interactions between different parts of the system are not always obvious, since they are not limited to direct physical connections. As a result, the impacts of changes are not always immediately apparent.

In principle it should be easy to detect dependences automatically, because the interface to each module is defined in what is called a "header file", which is explicitly referred to in all calling modules. Simply looking at these files overstates the dependences, however, since a module includes many routines, all of which are defined in the header file, but only a few of which may actually be used by the particular calling module. Furthermore, since it is sometimes time consuming to list exactly which other modules a module uses, programmers often use a file that simply defines everything that is likely to be necessary. Overuse of this file masks the real underlying dependences by (apparently) making everything depend on everything else.

Interactions can be determined directly from the source code of the system, for example, by looking for places where one module calls another. Cross-reference listings can be made that list which modules call which other modules. These listings exist and are used, but they have two limitations. First, the cross-reference does not indicate where modules use data structures from
another module (as opposed to calling routines). Second, the cross reference only covers the
can be of the kernel; it does not show which routines are called by code developed by other groups.
As a result of these problems, there seem to be no reliable mechanical means to determine
the interactions between different modules. Instead, social mechanisms are used. In theory, a
programmer should register with the document library if they want to use a documented
interface and ask the maintainer if they want to use an undocumented interface. An engineer
could then use these sources to determine who would be affected by changes to a module and
should be invited to review any changes. In practice, programmers sometimes borrow a
document or copy pieces of someone else's code and therefore do not realize that they should
inform the developer. In some cases, these other programmers are in other groups where the
usual social norms that control how interfaces are used may not apply.

4.1 Developing new forms

Given a flowchart of a process—such as could be easily generated from Table 2 above—a
common approach to redesign is to look for problems such as redundant or non-value added
steps or places where tasks spend long periods waiting to be worked on and then 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 are incapable of
doing anything to fix their problems. The process could be modified to allow customers do more,
for example, to check themselves for duplicate tasks. In fact, 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.
Our analysis suggests another approach to redesign, namely, replacing some coordination mechanisms with alternative mechanisms. In the remainder of this section, I will discuss three examples involving alternative techniques for managing task-task, resource-resource and task-resource dependences described above.

Alternative mechanisms for managing prerequisite dependences. In the problem fixing process, several activities appear to manage prerequisite dependences between tasks by ensuring that the output of the 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 actually fix the problem and do not introduce new problems. In addition to these tests, managers must approve changes before they can be implemented. Such approvals may serve as an additional check on the quality of the change, either directly, if the manager notices problems or indirectly, because 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, track how engineers spend their time or simply to demonstrate their power without necessarily adding any value at all. If approvals are a quality check, however, there are other mechanisms that might be appropriate. For example, if approvals are time consuming and most changes are approved, it may be more effective to continue the change process without waiting for the change to be approved. 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 dependences. Changes are problematic when they are visible outside a single module since they then require coordinated changes to the modules which depend on it. These dependences are especially difficult to manage if the modules are developed in different divisions, since there is little informal communication between divisions. For example, one interviewee told a story about the time the word processing system
became the source of mysterious system crashes. It turned out that its developers had used a very low level system call which had been changed between releases, causing the problem. There was no way for the developers of the word processor to know not to use the system call and no way for the developer of the system call to know they were using it, since the word processor is developed in another, geographically remote software development unit.

The typology suggests that such dependences must first be noticed and then managed. Noticing dependences is sometimes difficult, however. For example, engineers can only perform unit tests (that is, tests of a single module); they can not really test the whole operating system since they do not necessarily know how the other modules are supposed to behave or even have access to all the code. To save time, an engineer or even the integration group might not recompile all files, but since there are no sure-fire methods to determine which other modules might be affected by a change some affected files might be omitted, an almost certain source of problems.

One solution is to provide documented service interfaces for the data and calls programmers use. Such interfaces change only rarely, reducing the chance of problems. At the time of our study, the change control group was planning to control the use of more interfaces were planning a new set of interfaces to internal data that customers were using. A second solution is to better track what interfaces people are using. As mentioned earlier, there is a database that lists which engineers have requested copies of documents describing interfaces; in principle these lists can be used to track who uses a particular interface, but it is not clear how often they are used or how accurate they are. For example, engineers frequently borrow copies of documents or code fragments; these borrowings could result in dependences that are not captured by the database. One engineer was working on a database that included all currently known dependences, but he was concerned that without a mechanism for people to report their use of other modules the database would not stay up-to-date.
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 them to product engineers, who assign them to software engineers. In addition, software engineers may assign reports or subtasks to each other. Currently these assignments are done on the basis of specialization, that is, an actor with a problem report must determine the module in which the problem likely appears and assign the task to the engineer responsible for that module. This system has the advantage that a particular engineer is responsible for a small set of modules and can therefore develop expertise in that code. (This feature is particularly important when the engineer is also developing new versions of the system.) As well, 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 later transferred. A second problem is load balancing: one engineer might have many problems to work on while others have none.

Towards the end of our study, the group we were studying underwent a reorganization. The reorganization split the engineers into support and development groups, possibly to allow the development engineers to concentrate on adding new functionality. As each version of the system is made available to the customers, development stops and the version goes into support. Engineers working on these versions fix only important problems and refer others to the development engineers be addressed in future versions. Most bugs are reported against the current versions though, since those are the ones that customers have.

In this form, support engineers are not specialized by module, apparently because the support group has fewer engineers who do not split their time between support and development, thus reducing the need for specialization as well as the resources to support it. The
support engineers are instead organized around change ownership rather than module ownership, that is, an engineer is assigned a particular problem report and changes whatever modules are affected. As a result, task assignment can be done by workload rather than specialization.

With change ownership, multiple engineers may be working on the same module. To manage these new task dependences, the company started to use a source control system. The system maintains a copy of all source files. When engineers want to make a change to a file, they check it out of the library, preventing other programmers from modifying it. When the change 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.

*Insert Table 3 about here*

The previous example showed 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; engineers 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 lowest bidder is chosen and the task assigned to him or her.

4.2 Evaluating alternative organizational forms

Given this analysis, it is important to be able to evaluate the advantages and disadvantages of each kind of organization. In this section, I will suggest informally how such an evaluation might proceed for the three forms of task assignment consider above—specialist, generalist and market-like—following Malone and Smith's (1988) analysis of four different organizational structures. In their model, tasks arrive at an organization and must be assigned to an actor that can execute them. They compare organizational forms on three criteria: the
production cost (i.e., the average delay to process a task), the coordination cost (the number of messages necessary to assign a task) and vulnerability of the form to failures of an actor.

Many other factors could be added to complicate such a model. I will briefly consider three additional factors: 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 are 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 three messages, from the customer to the service centre, from the service centre to the marketing engineer and from the marketing engineer 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 will be able to fix problems relatively quickly once they get the 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 engineer is free, increasing the time to finish the task. The engineer does not have to wait for the code to become available, however. Also, other engineers may be under-employed, although presumably those engineers could be busy working on new versions of the system.

Finally, the form is vulnerable to the failure or overloading of a single actor since the engineer responsible for each module has no backup (in practice, of course, other engineers could substitute in a pinch). Assignment based on module reinforces specializations by module since engineers will have little opportunity or need to learn about other parts of the system.
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 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. The organization takes no advantage of any difference between actors in performance and no actor has much opportunity (or incentive) to learn in detail about a particular module to improve performance. 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, since 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, then 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 is 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 problem: if engineers are rewarded based 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.

*Insert Table 4 about here*
4.3 Effects of electronic media on the choice of coordination methods

As discussed above, the use of new communication media and other kinds of IT differentially affect the costs of coordination mechanisms. 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 group.) However, the database system provides much of the same functionality 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.

Rather than focusing on the specific technology then, one approach to analyzing such technologies is to consider which of a system’s attributes are important. Nass and Mason (1990, pp. 52–53) 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 be retrieved at a later date. Computer conferencing and databases have this property; telephones and ordinary electronic mail do not. 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 certain kinds of coordination. For example, such a system could filter problem reports for engineers based on an interest profiles, reducing the number that need to be evaluated. 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 dependences among tasks and resources. A variety of mechanisms are used to manage these dependences. For example, the possibility of duplicate tasks may simply be ignored or engineers may check for a known solution before attempting to solve the problem. Dependences 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 a variety of code management systems and disciplines.

The choice of coordination mechanisms to manage these dependences results in a variety of possible organizational forms, some already known (such as change ownership) and some novel (such a 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, thus reducing duplicate effort, or reduce the coordination costs of a market-like task assignment sufficiently to make it desirable.
As well, the software change process may have interesting parallels in other industries. Despite differences in the products, the other engineering change processes studied (Crowston, 1991) had similarities in 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 specialists. An analysis similar to the one presented here might reveal interesting alternatives in this domain as well. Such an effort may be particularly timely, given the leading role IT-enabled changes play in current proposals to revamp the American health care system.

Coordination theory, like all theories, is a simplification of the complexity of real organizations, but it seems to usefully explain a variety of alternative processes and highlight the contribution of new communications media and other information technologies. The single example presented here obviously does not serve to test the theory, but rather to demonstrate the potential of the approach. Given its focus on how tasks are performed, however, the technique may not appeal to those with other interests. Furthermore, the suggestions of the analysis needs to be tempered by consideration of omitted factors (as is true of any kind of analysis). Specifically, just because a particular mechanism is cheaper, does not mean it is automatically better or that it will or should be implemented. In other words, coordination theory does not make strong predictions about what should happen to any single organization that implements a new communication systems, although it does suggest what will happen in aggregate (Malone, et al., 1987). For example, 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 which an informed manager can consider and modify as appropriate for the particulars of the organization.

Therefore, the 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 dependences are managed and the implications of alternative
mechanisms. As an example, we are currently using these techniques to compile a handbook of processes (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 more precisely revealing 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 requires more rigorous methods for recording processes and identifying dependences in organizations. There are already many techniques for data collection which are relevant, but none focus explicitly on identifying dependences. 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 dependences 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 techniques indicate that it takes a while to pick up the concepts, but that using them leads to greater insight into the process.

Second, more work is needed to elaborate the typology of dependences, 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, that perhaps help realize the potential of electronic media and new organizational forms.
Acknowledgments

This research was supported by the Center for Coordination Science at the Massachusetts Institute of Technology and a fellowship from the Ameritech Foundation. It 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.
References


Draft of 8/17/94


### Table 1. A typology of dependences and associated coordination mechanisms from (Crowston, 1991).

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Coordination mechanisms to manage dependence</th>
<th>Coordination mechanisms to maintain dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task-task</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks share common output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>same characteristics</td>
<td>• look for duplicate tasks</td>
<td>• decompose a task into subtasks, including integration</td>
</tr>
<tr>
<td>overlapping</td>
<td>• merge tasks or pick one to do</td>
<td></td>
</tr>
<tr>
<td>conflicting</td>
<td>• negotiate a mutually agreeable result</td>
<td></td>
</tr>
<tr>
<td><strong>Tasks share common input</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shareable resource</td>
<td>• no conflict</td>
<td></td>
</tr>
<tr>
<td>reusable resource</td>
<td>• make conflict visible</td>
<td></td>
</tr>
<tr>
<td>non-reusable resource</td>
<td>• schedule use of the resource</td>
<td></td>
</tr>
<tr>
<td>Output of one task is input of other (prerequisite)</td>
<td>• pick one task to do</td>
<td></td>
</tr>
<tr>
<td>same characteristics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conflicting</td>
<td>• means-ends analysis</td>
<td></td>
</tr>
<tr>
<td><strong>Task-object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object is resource used by task</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• order tasks</td>
<td>• identify necessary resources</td>
</tr>
<tr>
<td></td>
<td>• ensure usability of output</td>
<td>• identify available resources</td>
</tr>
<tr>
<td></td>
<td>• manage flow of resources</td>
<td>• choose a particular set of resources</td>
</tr>
<tr>
<td></td>
<td>• reorder tasks to avoid conflict</td>
<td>• assign the resources</td>
</tr>
<tr>
<td></td>
<td>• add another task to repair conflict</td>
<td></td>
</tr>
<tr>
<td><strong>Object-object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One object depends on another</td>
<td>• identify the dependence</td>
<td>• pick objects with the desired relationship</td>
</tr>
<tr>
<td></td>
<td>• manage the dependence</td>
<td></td>
</tr>
<tr>
<td>Actor</td>
<td>Activity</td>
<td>Dependence managed between...</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Customer</td>
<td>find a problem while using system</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>report problem to response centre</td>
<td>problem fixing task and duplicate tasks</td>
</tr>
<tr>
<td>Response Centre</td>
<td>look for bug in database of known bugs; if found, return fix to customer and stop</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>attempt to resolve problem</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>refer hardware problems to field engineers</td>
<td>problem fixing task and duplicate tasks</td>
</tr>
<tr>
<td>Marketing Engineer</td>
<td>look for bug in database of known bugs; if found, return fix to customer</td>
<td>usability of problem report by next activity</td>
</tr>
<tr>
<td></td>
<td>request additional information if necessary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>attempt to reproduce the problem or find workaround</td>
<td>problem fixing task and actor’s time</td>
</tr>
<tr>
<td></td>
<td>if the report is actually a request for an enhancement, then treat it differently</td>
<td>task and resources required by tasks</td>
</tr>
<tr>
<td></td>
<td>determine affected module</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>if unable to diagnose, forward to SWAT Team</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if bug is in another product, forward to appropriate product manager</td>
<td></td>
</tr>
<tr>
<td></td>
<td>forward bug report to manager of group responsible for module</td>
<td></td>
</tr>
<tr>
<td>Programming Manager or Designate Software Engineer</td>
<td>determine engineer responsible for module and forward bug report to that engineer</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>pick the report with the highest priority, or the oldest, or the one you want to work on next</td>
<td>problem fixing task and actor’s time</td>
</tr>
<tr>
<td></td>
<td>diagnose the problem</td>
<td>task and resources required by tasks</td>
</tr>
<tr>
<td></td>
<td>if the problem is in another module, forward it to the engineer for that module</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>design a fix for the bug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>check if change is needed in other releases and make the change as needed</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>send the proposed fix to affected engineers for their comments; if the comments are negative, then revise the bug and repeat the process</td>
<td>two modules</td>
</tr>
<tr>
<td></td>
<td>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</td>
<td>management of usability task and capable actor</td>
</tr>
<tr>
<td></td>
<td>approve the change</td>
<td>usability of fix by next activity</td>
</tr>
<tr>
<td></td>
<td>write the code for the fix</td>
<td></td>
</tr>
<tr>
<td>Managers Software engineer</td>
<td>determine what changes are needed to other modules</td>
<td>task and subtasks needed to accomplish it</td>
</tr>
<tr>
<td></td>
<td>if necessary, ask the engineers responsible for the other modules to make any necessary changes</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>test the proposed fix</td>
<td>usability of fix by next activity</td>
</tr>
<tr>
<td></td>
<td>send the changed modules to the integration manager</td>
<td>task and capable actor</td>
</tr>
<tr>
<td></td>
<td>send the patch to the someone to send to the customer</td>
<td>usability of fix by integration activity</td>
</tr>
<tr>
<td>Integration</td>
<td>check that the change has been approved</td>
<td>usability of entire system by next activity</td>
</tr>
<tr>
<td></td>
<td>recompile the module and link it with the rest of the system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test the entire system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>release the new software</td>
<td></td>
</tr>
<tr>
<td>Agent</td>
<td>Activity</td>
<td>Dependence managed between...</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Customer</td>
<td>Use system, find a bug report bug to response centre</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td>Response Centre</td>
<td>lookup bug in database of known bugs; if found, return fix to customer and stop</td>
<td>problem fixing task and duplicate tasks</td>
</tr>
<tr>
<td></td>
<td>determine affected product and forward bug report to marketing engineer</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td>Marketing Engineer</td>
<td>lookup bug in database of known bugs; if found, return fix to customer and stop</td>
<td>problem fixing task and duplicate tasks</td>
</tr>
<tr>
<td></td>
<td>attempt to reproduce the bug—part of diagnosing it</td>
<td>problem fixing task and capable actor</td>
</tr>
<tr>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>Software Engineer</td>
<td>start work on the next bug in the queue</td>
<td>problem fixing task and actor’s time</td>
</tr>
<tr>
<td></td>
<td>diagnose the bug</td>
<td>management of usability task and capable actor</td>
</tr>
<tr>
<td></td>
<td>if it’s actually an enhancement request, then treat it differently</td>
<td></td>
</tr>
<tr>
<td></td>
<td>design a fix for the bug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td>Managers</td>
<td>approve the change</td>
<td>usability of fix by subsequent activities</td>
</tr>
<tr>
<td>Software engineer</td>
<td>check out the necessary modules; if someone else is working on them, then wait or negotiate to work concurrently</td>
<td>problem fixing task and other tasks using the same module</td>
</tr>
<tr>
<td></td>
<td>write the code for the fix</td>
<td>usability of fix by subsequent activities</td>
</tr>
<tr>
<td></td>
<td>test the proposed fix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>send the changed modules to the integration manager; check in the module</td>
<td></td>
</tr>
<tr>
<td>Integration</td>
<td>check that the change has been approved</td>
<td>usability of fix by subsequent activities</td>
</tr>
<tr>
<td></td>
<td>recompile the module and link it with the rest of the system</td>
<td>integration</td>
</tr>
<tr>
<td></td>
<td>test the entire system</td>
<td>usability of entire system by next activity</td>
</tr>
<tr>
<td></td>
<td>release the new software</td>
<td></td>
</tr>
</tbody>
</table>
Table 4. Relative costs of different task assignment mechanisms.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Specialists</th>
<th>Generalists</th>
<th>Market-like</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Production costs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Waiting for engineer</td>
<td>Necessary</td>
<td>Unnecessary</td>
<td>Unnecessary</td>
</tr>
<tr>
<td>Waiting for module</td>
<td>Unnecessary</td>
<td>Necessary</td>
<td>Necessary</td>
</tr>
<tr>
<td>Fixing problem</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Takes advantage of learning</td>
<td>On assigned modules</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Coordination costs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagnoses</td>
<td>3</td>
<td>2</td>
<td>2+N</td>
</tr>
<tr>
<td>Messages to assign</td>
<td>3</td>
<td>3</td>
<td>2N</td>
</tr>
<tr>
<td>Decomposition and assignment of subtasks</td>
<td>Necessary</td>
<td>Unnecessary</td>
<td>Unnecessary</td>
</tr>
<tr>
<td><strong>Vulnerability to failure</strong></td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Note: N is the number of software engineers.
<table>
<thead>
<tr>
<th>Date Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 16 1996</td>
</tr>
<tr>
<td>Aug 3 1996</td>
</tr>
<tr>
<td>Nov 1 1996</td>
</tr>
<tr>
<td>Jul 29 1996</td>
</tr>
<tr>
<td>Sep 3 0 1999</td>
</tr>
</tbody>
</table>

Lib-26-67