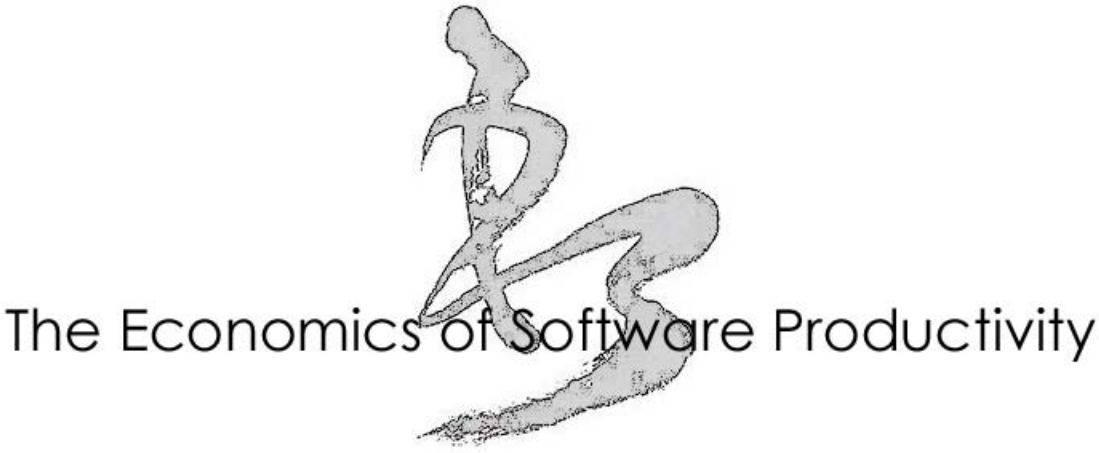


Yeong Zann Seow
(Advisor: Professor Kathryn Shaw)
23 April 2003

Abstract

The successful Systems Analyst must have an acute knowledge of productivity and the means by which productivity may be increased in his firm. The aim of this paper is to explore a few of the factors that affect productivity and in doing so, provide the manager with an understanding of how productivity can be increased in an organization. It proposes a Software Productivity regression equation, relating Productivity (the output) and various business factors.

1 Introduction

As Software Development firms prepare to compete in the e-business era, the ability to produce quality software on time is emerging as an important source of competitive advantage. Because software development is notoriously slow, firms have been experimenting with new approaches to software application development to meet current business needs. In response to these needs, firms have introduced quality management approaches, automated software design and development tools, and process improvement initiatives to their traditional methodologies [30]. In spite of these efforts, the software industry is still plagued by poor schedule, cost, and quality numbers [21]. The reason behind this could be due to either the true efficacy of the methods and tools or the organizational challenges in successful adoption of these proposed solutions or both.

A productive company would be able to complete more software projects in a shorter time, often producing them with fewer defects. In Economic sense, this translates into higher output and higher revenue, thus resulting in higher profit. Where the industry is highly competitive, productivity could be the deciding factor between the sustainability and failure of a Software company. Furthermore, due to the time-sensitiveness of the Industry, the speed at which a company is able to respond and fulfill a client's order impacts many business decisions today. There is an ever present threat to the firm - of market share eroding with time - as customers will not wait for a delayed product when alternatives are readily available. For example, Ashton Tate a former database software leader lost its rein on the market due to a delay in releasing its (then) latest version dBase. Unnecessary long software development times may also indirectly affect other products produced by the company. A prominent example is IBM's late launch of its PowerPC machines due to software delays. Lesser known examples are the numerous electronic manufacturing firms in Japan who, due to software delays, deliver their electronic products late, or with less features than intended - resulting in dissatisfied consumers.

Under traditional economics, software programmers are utility maximizers. They are primarily motivated by self-interest, and they value both leisure and the goods and services money can buy. Likewise, they seek to avoid unpleasant or otherwise costly activities. 'Putting forth their best efforts' may entail working when they do not feel like it or engaging in activities that, other things equal, they would rather avoid. How can employers make the programmers more efficient? What policies can be devised to induce a high level of effort from each programmer?

A textbook approach to this question would be to supervise the workers and to increase their pay. Ehrenberg and Smith writes about supervision:

One way to motivate high levels of effort is to closely supervise employees. While virtually all employees work under some form of supervision, close and detailed supervision is costly. Tasks in any production process are divided so that the economies afforded by specialization are possible. In all but the most manual, repetitive tasks, the worker must make decisions or adjustments in response to changing conditions. To insist on extremely close supervision would mean that the supervisor must have all the

information exactly the same time as the worker – in which case the supervisor might just as well make the decision! In short, detailed supervision destroys the advantages of specialization…

These problems of supervision are particularly relevant in the Software Industry. Constant supervision is simply an impractical approach.

The other conventional method is motivating workers through pay. From the employer's perspective, the big advantage of individually based output pay is that it induces employees to adopt a set of work goals that are consistent with those of their employer. Employees paid a piece rate are motivated to work quickly, while those paid by commission are induced to very thoroughly evaluate the needs of the firm's customers. Moreover, these inducements exist without the need for, and expense of, close monitoring by the firm's supervisors.

However, incentive based schemes has its own problems. For one, it is difficult to satisfy both employer and employee. Basing one's pay on one's current output places employees at risk of having earnings that are variable over time. External factors such as illness, hardware failure will adversely affect the employee's earnings if an incentive pay based scheme is solely used. The programmers may not be willing to bear the risks associated with such a pay scheme.

Another problem is the extent at which employee's measurable effort and the employer's objective are correlated. Imperfectly designed performance measures run the risk of inducing employees to emphasize that part of the performance that is measurable and to ignore the other aspects. In the Software Industry, the most commonly used measure of output is LOC (lines of code).

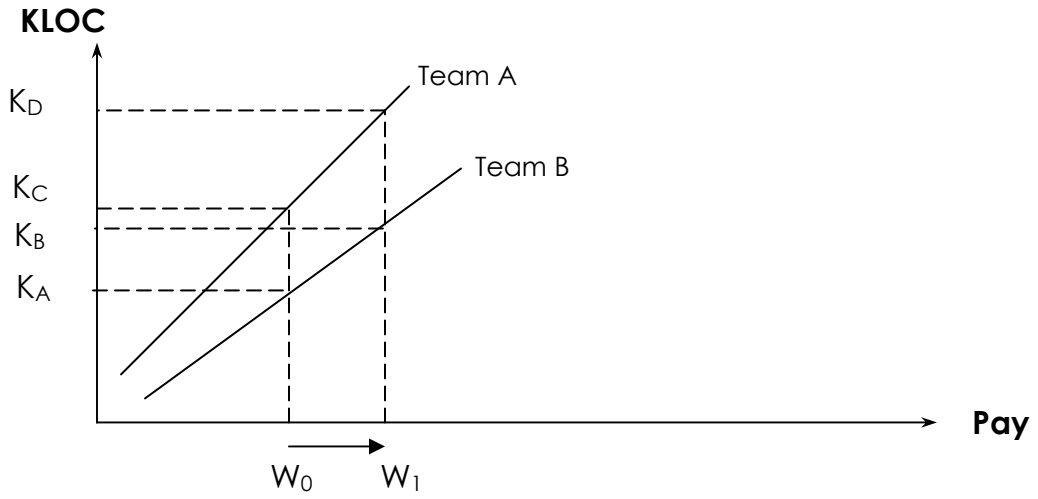


Figure 0-1a: Illustrating the effects of an incentive-based scheme on LOC

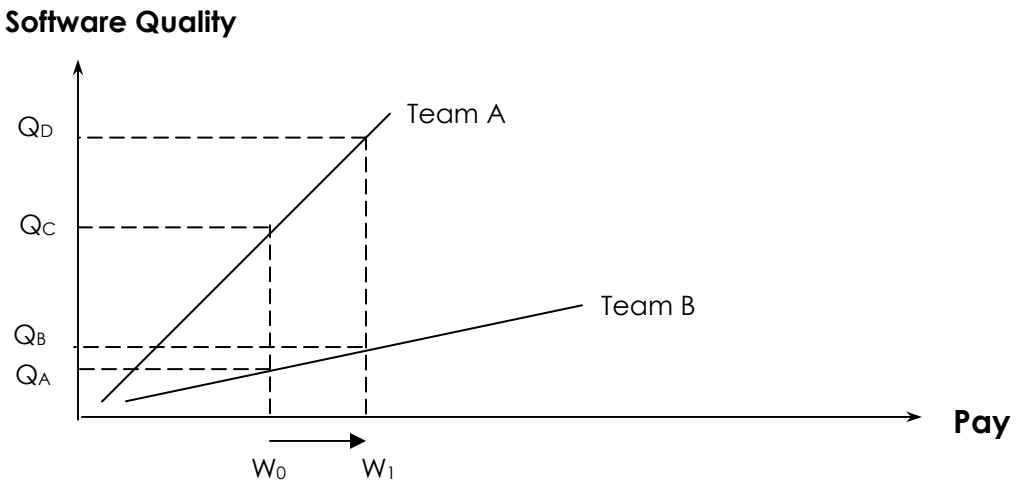


Figure 0-1b: Illustrating the effects of an incentive-based scheme on Quality

Increasing the employee's wages may serve to increase the output which is measurable and quantifiable, but this may be different from what the employer hopes to achieve. In our case, the measurable output is KLOC (thousands of lines of code) and the firm's ultimate objective is to increase Software Quality. Figures 0-1a and 0-1b illustrates this scenario. Take for example, two hypothetical programming teams Team A and Team B which are paid based on their output (KLOC). Increasing their wage rate from W_0 to W_1 will cause their output to increase. For Team A, this increase is from K_c to K_d and for Team B, this increase is from K_a to K_b. This increase follows from the conventional economics supply and demand theory, in which we treat ***KLOC*** as the output and ***pay*** as the price for that output.

However, due to certain factors (such as the nature of the programming task, or other human factors) the correlation between KLOC and Software Quality for Team A is stronger than for Team B. This results in Team B's software quality increasing by proportionately less, given the same increase in the wage-rate. In other words:

$$\frac{\%\ \text{Increase in Quality for Team A}}{\%\ \text{Increase in Quantity for Team A}} > \frac{\%\ \text{Increase in Quality for Team B}}{\%\ \text{Increase in Quantity for Team B}}$$

This issue is more complicated than it seems. Software Quality cannot usually be determined immediately upon delivery, it has to undergo a complicated and rigorous testing procedure before its strength can be determined. Logic errors and bugs may only be apparent after subjecting the software through testing. Thus with no obvious metric on Software Quality, the firm can only measure that which is clearly visible – the number of lines of code.

The firm's real objective in raising Wages from W_0 to W_1 is for a notable increase in Software Quality. A weak correlation between Quality and Quantity undermines this aim.

Other problems with an incentive-based pay scheme include difficulties of setting the pay rate (due to the specific software requirements of each project, how can we determine a rational pay rate) and difficulties of quantifying individual output in a group project.

These problems suggest that wages alone are not sufficient to effectively increase Software Productivity. This paper will be focused on two areas: finding a practical measure of Software Productivity and researching the means by which a manager can improve productivity in the firm.

2 Factors affecting Producivity

FACTOR F_1. TEAM SIZE (COMMUNICATIONS)

In a study conducted by Jones [5,6], it was discovered that any step towards the use of structured techniques, interactive development, inspections etc. can improve productivity by up to 25%. Use of these techniques in combination can yield improvements of up to 25% t0 50%. The only single technique that by itself can improve the programming level by more than 50% is a change in programming language. Yet higher gains can be achieved by highly skilled programmers, or teams of highly skilled programmers. Gains of more than 100% can be achieved by database user languages, application generators and software reuse (see previous factor). While these gains appear to be enticing, it was also discovered that certain adverse factors (known as 'Dominators') can suppress the effects of these other good factors. These 'Dominators' can reduce software productivity by an order of magnitude.

Communications is one of the dominators of productivity [S1]. If there are n workers working on a software project, there are n(n-1)/2 pairs of distinct communication channels within them. As n increases, the number of channels grows exponentially, and the increased time spent on communications can have a major influence (negative) on productivity.

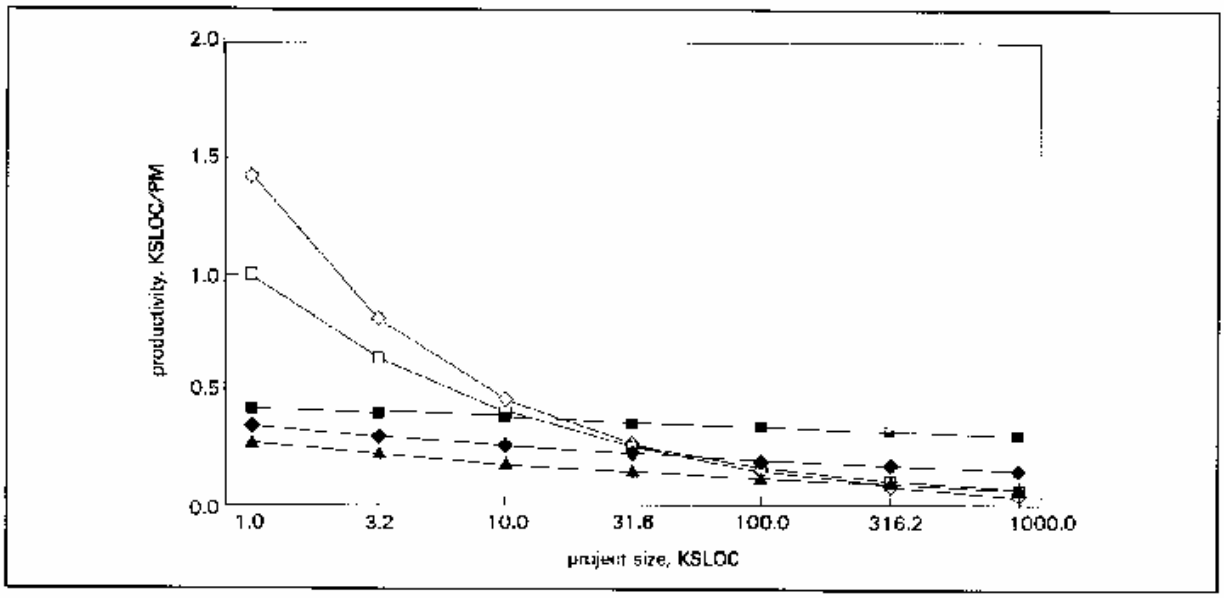


Fig. 1 Productivity versus project size

■ COCOMO organic [30] ◆ COCOMO: semi detached [30] ▲ COCOMO: embedded [20] □ Jones [6] ◇ Halstead [21]

Productivity studies have shown that as the Software Project Size increase, the productivity (measured as the ratio of output product divided by the input effort to

produce that output: Lines of Code / Person-Month). In practice, input effort is largely based on estimation and is thus subjective. The different graphs above exhibit variations in productivity due to software complexity. The 3 different complexity levels: COCOMO organic, COCOMO Semi-detached and Embedded Mode can be predicted by Boehm's COCOMO equations [20]. For all three cases, it can be observed that the law of diminishing returns to scale hold true.

Table 1 Effort equations

Reference	Effort in person-months
COCOMO: organic mode [20]	$2.4\ (KSLOC)^{1.05}$
COCOMO: semi-detached mode [20]	$3.0\ (KSLOC)^{1.12}$
COCOMO: embedded mode [20]	$3.6\ (KSLOC)^{1.20}$
Jones [6]	$1.0\ (KSLOC)^{1.40}$
Halstead [21]	$0.7\ (KSLOC)^{1.50}$

Table 2 Productivity equations

Reference	Productivity in KSLOC/PM
COCOMO: organic mode [20]	$0.42\ (KSLOC)^{-0.05}$
COCOMO: semi-detached mode [20]	$0.33\ (KSLOC)^{-0.12}$
COCOMO: embedded mode [20]	$0.28\ (KSLOC)^{-0.20}$
Jones [6]	$1.0\ (KSLOC)^{-0.40}$
Halstead [21]	$1.43\ (KSLOC)^{-0.50}$

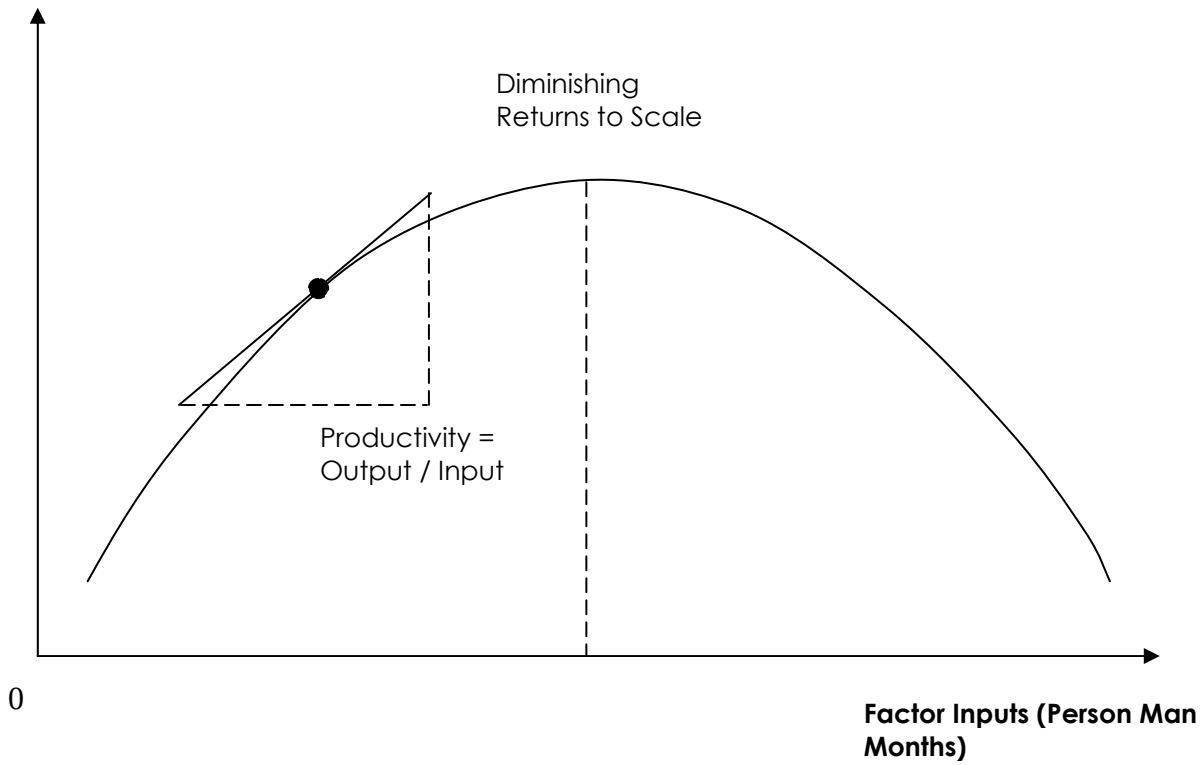
We can assume that a system of one KSLOC (thousand source lines of code) would be developed by a small group or an individual programmer. A project of thousand KSLOCs would be developed by large organizations composed of many developers. We will examine the extent communications influence these productivity statistics.

EFFECTS OF INCREASING PROGRAMMING TEAM SIZE

Individual programmers working alone do not have interruption from group members. There is no need to divide the programming tasks and thus the ratio of productive time to total time spent can be large, for motivated programmers. Esterling [30] states that one programmer working 60 hours a week can complete a project in the same time as 2 other programmers working normal hours, but at three quarters of the cost. If overtime is not paid, then costs are halved.

Small groups of experienced programmers can create large systems. Pyburn Systems scours the US for the best analytical thinkers, who love to create order from chaos, love to work long hours and are driven to succeed [31]. They work in small teams of less than 5 people to produce large software systems. A high pressure environment, small team and long hours often produces the best results.

As the size of a software project grows, the number of programmers needed increases and communication will tend to dominate productivity. Jeffrey [32] asserts that there is a point where coordination overheads outweigh any benefits that can be obtained by the addition of further staff. The dramatic increase in effort needed as the size of the software systems grows is the result of the difficulty in coordinating the large number of programmers who are trying to perform the tasks in parallel. This is coherent with the economic theory of diminishing returns to scale. There exists a point in which additional input factors do not have any more beneficial effects on the production function.



The graph above shows the theory of diminishing returns to scale. The slope of the curve represents productivity, and declines as we increase the factor inputs to the production function. At some point, the productivity becomes zero and after which increasing the number of team members yields negative productivity.

A software group communications model was developed that simulated the actual communication links between groups [11]. The simulation demonstrated that, due to programmer communications, group productivity is affected by organizational structure; there is an upper limit to the number of programmers that can effectively be added to a group to increase group productivity; and the potential of highly productive people can be neutralized by assigning them to positions with high communication requirements.

Jones pointed out that this view was explored initially in the 19th century by the pioneering organizational researcher Graicunias [33, 34], whose work on military organizations introduced squads and platoons into the US Army. The key observation was the geometric increase n(n-1)/2 of communication channels as n increases, and this research led to the conclusion that the upper limit of effective staff size for cooperative projects was about 8.

FINDINGS

Software developers have many duties that occupy their time during a typical work day. A highly productive developer spends 51% to 79% of a day productively working on software development [11]. We assume that software developers spend 65% of a 8 hour work day as productive time. Esterling [30] found the average duration of work interruption was 5 minutes a day for the average programmer. 2 minutes is needed to regain the train of thought, making the average total time spent on an interruption to be 7 minutes. In 5 hours of productive work (65% of 8 hours) a day, each interruption takes 2.33% of the productive time, 10 interruptions would cost 23.3% and 20 interruptions (highly excessive) would cost 50% of the productive time.

Simmons [1] examined the effect of only a few interactions between group members:

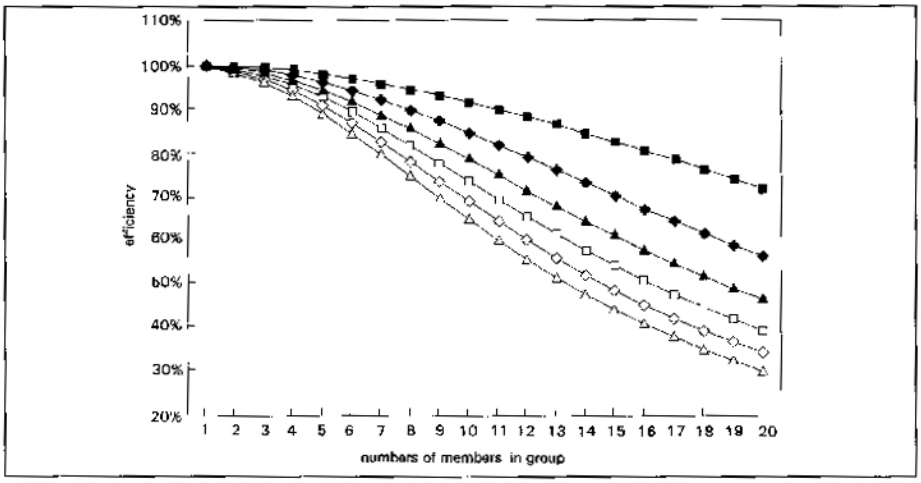


Fig. 5 Efficiency versus group size with few interactions where $f_s = 0$

■ $f_c = 0.1\%$ ◆ $f_c = 0.2\%$ ▲ $f_c = 0.3\%$ □ $f_c = 0.4\%$ ◇ $f_c = 0.5\%$ △ $f_c = 0.6\%$

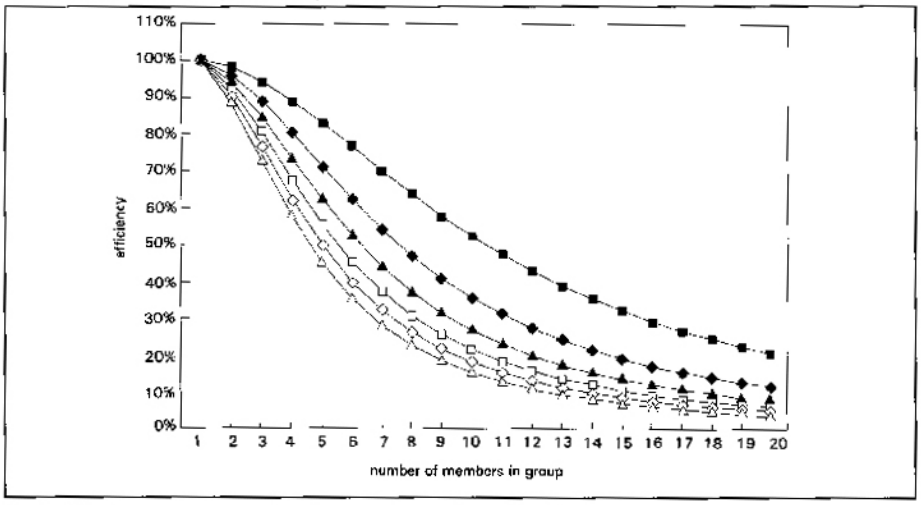


Fig. 6 Efficiency versus group size with many interactions where $f_s = 0$

■ $f_c = 0.1\%$ ◆ $f_c = 0.2\%$ ▲ $f_c = 0.3\%$ □ $f_c = 0.4\%$ ◇ $f_c = 0.5\%$ △ $f_c = 0.6\%$

In the figures shown above, efficiency was computer for f_s equal to zero and f_c between 0.1% to 0.6%, where f_c is the portion of total time spent interacting with each of the other group members. For these values, a group of eight members would be between 75%

and 95% efficient. In the lower table, efficiency was computed for values between 1% and 6%, which represents many more interactions between group members. For the same group of eight members, the efficiency would range from 23% to 64%. As the channels of communications continue to increase, efficiency would continue to decrease.

Group efficiency is related to the group productivity speed up ratio R_n , which expresses group productivity in terms of the productivity of a 1-member group.

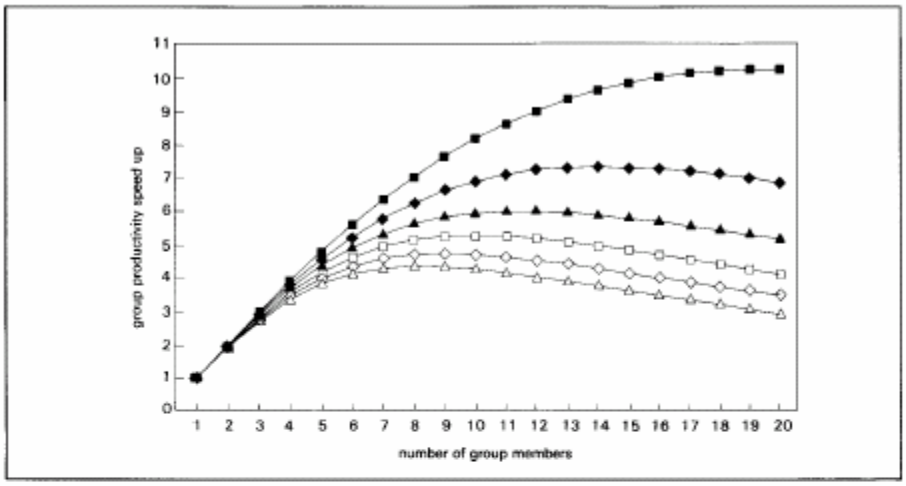


Fig. 7 Group productivity speed up for normal interactions and $f_e = 0$
■ $f_c = 0.25\%$ ◆ $f_c = 0.50\%$ ▲ $f_c = 0.75\%$ □ $f_c = 1.00\%$ ◇ $f_c = 1.25\%$ △ $f_c = 1.50\%$

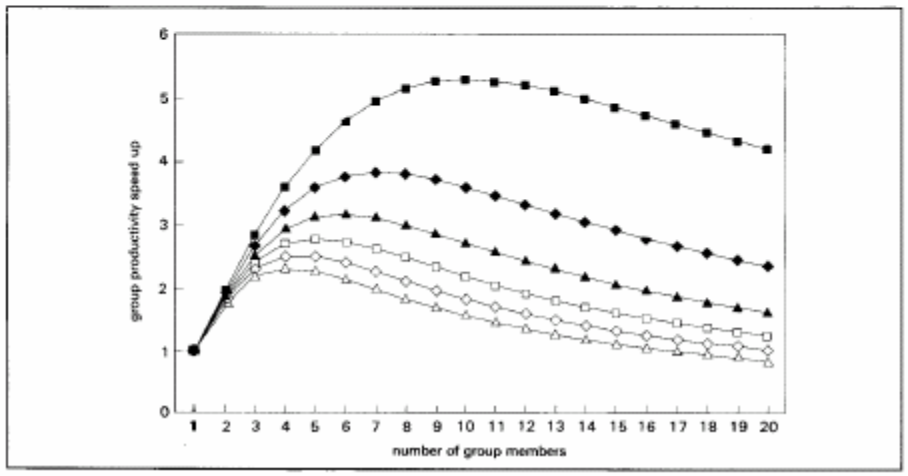


Fig. 8 Group productivity speed up for excessive interactions and $f_e = 0$
■ $f_c = 1\%$ ◆ $f_c = 2\%$ ▲ $f_c = 3\%$ □ $f_c = 4\%$ ◇ $f_c = 5\%$ △ $f_c = 6\%$

In the figures above, group productivity speed up is shown for fc that ranges from 0.25% to 1.50%. Speed up of a 5-member group would be between 3.8 and 4.8, an eight-member group would be between 4.3 and 7.0, and a 20-member group would be between 3.0 and 10.3.

Note that, for a given f_c, as the number of group members increases, the speed up increases, peaks and falls off. For an f_c of 1.0% the speed up levels off at eight group members. For an f_c of 1.5%, there would be no reason to have more than six group

members. As the interactions between all group members become excessive, the speed up declines dramatically, as can be seen in Fig. 8. As the f_C varies from 3% to 6%, the optimum group size varies from five members to two members.

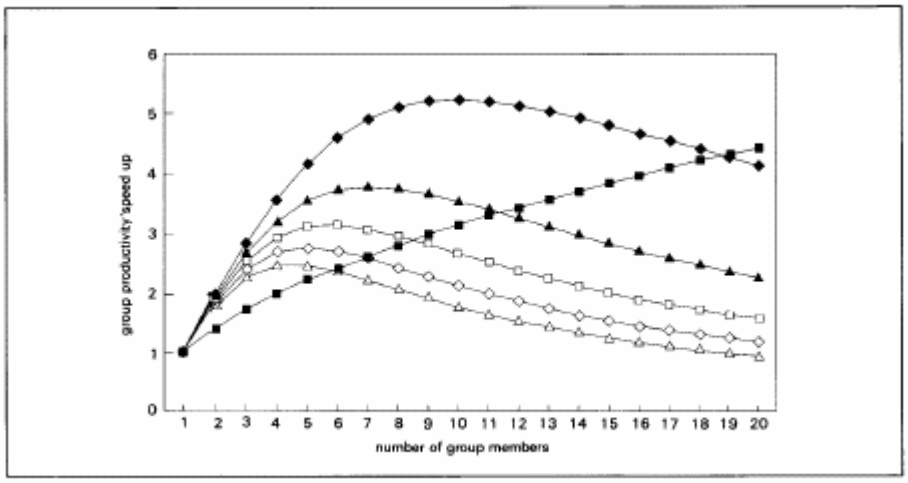


Fig. 9 Conte model [18] and communications model (i.e. $f_c = 0$) speed up comparison
■ Conte model ◆ $f_c = 1\%$ ▲ $f_c = 2\%$ □ $f_c = 3\%$ ◇ $f_c = 4\%$ △ $f_c = 5\%$

In the figure above, the speed up calculated from the 187 database of Conte et al [18] is compared to speed up where communication varies. Note that the speed up based on Conte's productivity equation, when plotted as a function of group size, is an entirely different shape to the communication speed up curves.

CONCLUSIONS

Excessive intra-group interactions result in communications dominating productivity. Both designers and managers affect the number of interactions. A poor design causes more interactions than a good design, and a good design causes more than a great design. Managers determine a scheduled time to complete a project, the number and experience of developers assigned to a project, and the organization of the developers into groups. The potential of highly productive programmers can be offset by assigning them to positions requiring high communications. When the size of a group exceeds 8 members, communications begin to dominate productivity and cause group productivity to decline as additional group members are added. Thus, the optimum group size for a single software development is between 5 to 8 members.

FACTOR F_2. COLLOCATING TEAM MEMBERS

MOTIVATION FOR COLLOCATION: SUPERIOR COMMUNICATION

As can be seen from the point above, communications take a large portion of an individual programmer's time. Past studies have indicated that less than 30 percent of a software development programmer's time in large projects is spent on traditional programming tasks and less than 20 percent of the time is spent on coding [6]. The rest of the programmer's time is spent in design meetings, resolving problems within the team, resolving specification misunderstanding with the customers, other communications with the customer and product testing, etc. Breakdowns in communication among development team members and with the customers often cause schedule delays, especially from rework when the delivered system is not fitting the users' needs. In addition, unplanned interruptions constitute a significant time sink, as does the time lost in context switching [36]. In traditional software development, formal artifacts and documents produced by one group are assumed to be sufficient to provide all of the information needed by another group. However, this is often not true [13]. The fact that tasks such as design, coding, systems, and integration testing are carried out by different groups hinders communication of both technical information as well as its rationale, resulting in project delay and rework. Even the physical separation of the software development team within a building can create several forms of communication breakdowns. Specifically, there is a logarithmic decline in communication with increased distance between collaborators, where any distance over 30 meters produced the same low probability that team members would talk to one another [4], [28]. Numerous ethnographic studies of teamwork reveal the subtle, hidden nature of the features that make communication effective. For example, chapters by Suchman, Hutchins and Klausen, and Heath and Luff in Engestrom and Middleton [14] examined transcripts of conversation from work practices to show how group work and workspaces are mutually constituted. Team members systematically communicate information both through various posted messages, their glances, the arrangement of chairs and desk, etc., to help accomplish the group task. Hutchins [22] has proposed a theory of "distributed cognition" to provide a theoretical framework for understanding how people exploit features of the social and physical world as resources for accomplishing a task (also termed "socially shared cognition" in [38] or "situativity theory" in [19]). In this view, communication creates mutually held representations of the work that allow activity to proceed successfully within a complex and ever changing context [23], [46]. The concept of common ground is used to describe how conversations proceed by creating shared understanding between participants [11]. This shared understanding is essential to conducting any joint activity. Communication breakdowns occur in a number of ways. For example, members of design teams can occasionally mistakenly assume that the others share a common understanding of an issue when in fact they do not. These confusions usually arise when each team member starts with an unstated assumption and is not able to immediately resolve the problem once a conflict is detected. When the team members are physically separated, this error will often not be detected until the following design review meeting when the group meets face-to-face. Meanwhile, other members of the team would have continued their work assuming that there was no error. A design change to rectify this mistake at a later stage will further add to the project delay. Such situations are not uncommon in any large software project. The same phenomenon is also observed when there is a communication breakdown between developers and the customers of the project. Ambiguity of customer specifications and misunderstandings of specifications are prevalent in customized application development. Once again, these problems can be attributed to conflicting unstated

assumptions that are not resolved immediately. There is evidence suggesting that collocation of the project team and customers in a war room can be effective in reducing such communication breakdowns and facilitating speedy resolution of conflicts. By improving communication, productivity and timeliness of the projects will also improve.

TEAM COLLOCATION THROUGH "WAR ROOMS"

In corporate America, collocation is achieved in what are called "war rooms." The term generally refers to an immersive environment where experts, technology, managers, and new products come together in a "nerve center" to facilitate interactive information sharing with a minimum of outside distraction. In this context, the war room connotes the centralization of all the best resources into one location to promote efficiency and timely work output. These dedicated project rooms have also been called "skunk works" [43] or "team rooms" [45]. There have even been several software implementations of "virtual war rooms" [18] in an attempt to achieve these same ends while being remote, a point we will return to in the discussion. Driving the popularity and perceived significance of war rooms are several factors. First, it is economical to locate people in the same place at the same time. Although today's technology provides workers with a growing number of tools for interacting over distance (e.g., e-mail, message pagers, instant message systems, videoconferences), there is an enduring preference for face-to-face interactions [10]. Further, with collocation, every team member can be aware of all aspects of the project development without the need for scheduling status meetings and circulating written progress reports. Finally, corporate culture associates the worker's status with size of a workspace, proximity to high status co-workers, and other locally relevant perceived advantages of a workspace [12]. Therefore, projects given a dedicated space are seen by outsiders as places for high intensity, important activities. Team members selected to participate in war room projects are usually hand picked by managers because of their particular skills and the perceived importance of the project over the worker's routine tasks. Several evaluations of workspaces lead us to hypothesize that collocating members of software development teams will enhance productivity. In an investigation of integrated product teams, Poltrock and Englebeck [37] described how physical collocation facilitated collaboration and coordination within teams via both scheduled meeting and opportunistic interactions. Sawyer and his colleagues found that team rooms helped focus the activities of the work group and isolated them from interruption from people outside the project [43]. Becker and Steele at Cornell's International Workplace studies program surveyed a number of case studies on collaborative teams and found that the way an office environment is organized influenced work processes such as coordination, work patterns, and communication internal and external to the team [8]. Kraut et al. [28] studied collaboration between scientific researchers and found that the physical distance between offices influenced the development of collaborative relationships and the execution of the work. Allen's [4] investigation into communication patterns in R&D laboratories found that engineers are more likely to communicate with the individuals nearest to them and that people tend to communicate more with people from the same group than from other groups. Although these studies provide evidence that war rooms should increase communication and facilitate efficient flow of work, they lack formal indicators of measurable performance outcomes of a project. Some of these communication-related issues are resolved partly in the various approaches to systems development discussed above. However, we believe that collocation of the entire team in a war room (with its concomitant limiting of the scope of the project) can combine several advantages to overcome the breakdowns. For example, collocating the

customer with the designers and developers brings in the advantages of quick customer feedback and fast resolution of requirement questions, as in prototyping and joint application development. Collocating the team also helps in resolving misunderstanding among designers and developers and also helps to ensure adherence to formal procedures and quality standards, as in the traditional waterfall model. Although collocation will increase interruptions, the interruptions are about the project itself, producing only minimal loss from context switching.

EXPERIMENT

A research conducted by Teasley, Covi, and Olson [3], at the Rapid Software Development Center (RSDC), (consisting of six war rooms, additional conference rooms, and hotelling2 areas) strived to study collocation by implementing the following measures:

(1) Productivity indicators, standard measures including time to market, and function points per staff month,

(2) Questionnaires, administered at the beginning, asking all team members about their predictions about their satisfaction with the facilities and again, at the end of the project, assessing their actual satisfaction with the facilities,

(3) Observations of two teams in depth from visits with them about 8-10 hours a week for the duration of the projects and interviewing the team members at project completion,

(4) Questionnaires, administered at project completion, assessing team satisfaction, customer satisfaction, and sponsor satisfaction for all pilot projects. These measures are described in detail below.

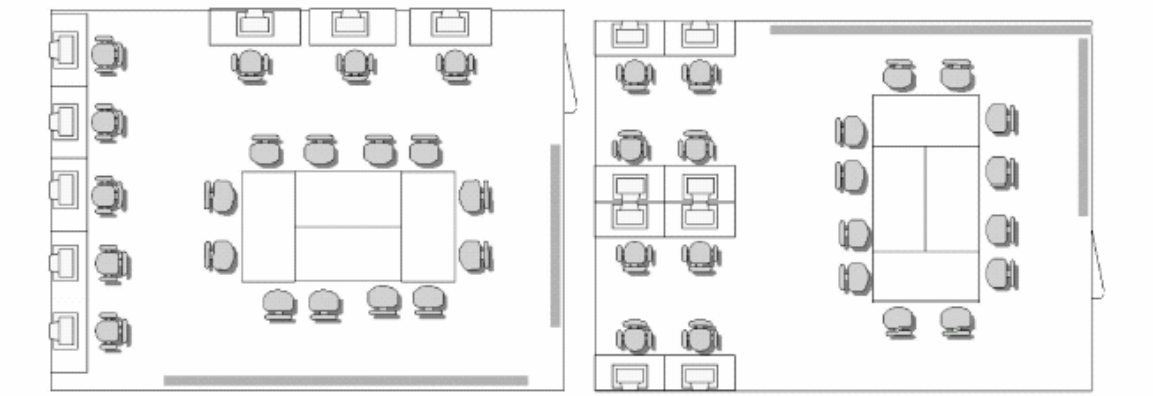


FIGURE 2-1

The facilities at the RSDC included a dedicated war room for each software development team, conference rooms nearby, and various hotelling cubicles for more private work away from the team. Fig. 2-1 shows the general layout of the rooms. The dedicated war rooms were outfitted with individual workstations for each of the team members. Workstations were arrayed along the outside walls, shown in the lefthand panel of Fig. 2-1, or in a "E" shape, shown in the righthand panel of Fig. 2-1, more like individual cubicles, but without any walls. In the middle of the room was a worktable, the walls had whiteboards, and flip charts on easels were available as needed. Several

rooms had printing whiteboards. Near the war rooms were conference rooms, available on a first-come-first-served basis. One conference room was outfitted with video conferencing for use in remote meetings with others as needed. "Hotelling" space–unassigned, more private, quiet cubicles with workstations and phones–was also available nearby. These facilities contrast with the company standard of relatively large individual cubicles with 6-foot walls near cubicles with people who may or may not be associated with the same project. The company benchmark statistics come from people in such cubicles.

The Teams

The six teams ranged in size from six to eight people. Each team consisted of a manager, three to four contract employees for the programming, and two to three business partners from within the company who were the intended (internal) customers of the applications. The teams shared the services of methodologists, technical architects, database experts, and testing specialists. For the duration of the RSDC project, the team members were not sharing their time with any other projects. The teams in this study differ from the teams included in the company benchmark statistics on only these factors. They were neither individually selected for the teams nor newly hired for the experiment.

The RSDC Projects

The six projects selected as pilots for the RSDC were developed on client-server and mainframe platforms. The projects came from several areas of the company, including manufacturing, finance, market and sales systems, and purchasing. The projects included a global financial database, a system to measure equipment effectiveness, a retail website, a system to support competitive analysis, a bill of material audit, and a database. The projects ranged in size from 326 to 880 function points (a standard measure of size described in Section 3.5).

Measures of Team Experience

Questionnaires were used to measure the teams' experience with and preferences for various kinds of workspaces and tools. The entrance questionnaire had 103 items and took about 20 minutes to fill out. This questionnaire asked about the person's prior experience with various facilities and technologies using a 5-point scale anchored with "not at all" to "very frequently." This questionnaire assessed team members' predictions of how frequently they would use new facilities like war rooms and hotelling space, conference rooms, etc., at the RSDC, as well as new technologies available to them at the RSDC. The questionnaire also asked team members to assess how well they liked to work in various facilities or with various tools, using a modified 5-point Likert4 scale with "strongly dislike" to "strongly like" as the anchors. The questionnaire also asked for team members to assess their preferred work styles, again using 5-point Likert scales, with "strongly disagree" to "strongly agree" anchors. The latter items are listed in Table 1. Nine items, starred in the table, are those intended to assess the kinds of preferences that people have about characteristics we thought the new facilities would engender, such as being busy and collaborative. The exit questionnaire contained 71 items of similar content as in the entrance questionnaire. However, instead of predicting the future frequency and preference, these asked how often team members did use various spaces and technologies and how they liked or disliked them. This questionnaire also asked the same questions shown in Table 1. To better understand the reasons behind their opinions, all the team members from two of the pilot teams were interviewed

individually. These team members were encouraged to talk freely about the advantages and disadvantages of the war rooms, the aspects of team dynamics, their attitudes about the layout and equipment in the rooms, why and when they used hotelling space, and other features they would change to make the RSDC a better environment. Interviews took about one hour to 1 1/2 hours. For the two teams studied in-depth, each individual was asked to fill out a short 9-item questionnaire biweekly, asking both for some quantitative assessment of the team and war rooms, but also some narrative responses to back up their numerical responses, answering in each case "Why or why not?" The same two teams were observed for about 8-10 hours over the course of the projects. Observation notes were transcribed immediately after each observation session and then clustered by two of the researchers into major categories of actions/statements relating to the research question.

RESULTS

The productivity of the pilot teams was compared with two benchmarks, shown in Table 2. The Function Points/Staff Month and the Cycle Time to both the industry standard for projects of comparable sizes and the baseline for the company was compared. The company baseline numbers for productivity and cycle time were computed in the following way: A consulting firm that specializes in software metrics was hired to select a sample of past software projects from multiple domains and functional areas within the organization. No outlier projects, i.e., extremely large in size or with long duration, were selected. Ninety-three projects were selected from different functional areas in the organization and the productivity and cycle time numbers of these projects were used to arrive at the company baseline measures. The sample of projects were divided into two subsamples based on the two major platforms for development, i.e., mainframe and client-server used in the organization. The baseline measures for mainframe and client-server projects were computed using simple average of the data collected from these projects. Since the software process used prior to RSDC involved all the stages in the waterfall model and the RSDC projects started with a well-documented design, we adjusted the baseline productivity and cycle time numbers to include only the stages covered in RSDC.5

TABLE 1
The 16 Items Used in the Assessment of Workstyle

1. *I prefer to work independently as much as possible.
2. *I am easily distracted by activities taking place in the same room as me.
3. I would like to work in a paperless office.
4. People working on special projects need to stay connected to their reporting manager.
5. I prefer reading from print on paper than reading on the screen.
6. *Regular meetings are necessary for my work.
7. I would like to personalize my workspace (e.g. photographs, personal items).
8. I like working on tasks defined by others.
9. *I like to collaborate as much as possible.
10. *I think that it is important for people on special projects to work near each other.
11. *I like working in a busy environment.
12. I would like to work at different locations during the day.
13. *I am concerned about how my contribution to the project will be recognized.
14. I prefer participating in defining the tasks I work on.
15. *Running into people I know in the hallway is a convenient way of interacting.
16. *Technology will replace the need for people in the same project to work near each other.

TABLE 2
Comparative Statistics on Productivity Measures

	Pilot Teams	Company Baseline	Industry Standard
FP/Staff Month *(higher is better)*	29.49 (*SD* = 7.88)	14.18	11.25
Cycle Time per 1000 FPs *(lower is better)*	7.64 (*SD* = 3.37)	12.74	11.75

The industry standard numbers were obtained from the SPR database after adjusting for the size of the projects used in our sample [26]. SPR presents industry benchmarks for projects implemented on mainframe and client-server platforms. These benchmarks are presented for project sizes ranging from 200 to over 10,000 function points and are also adjusted to include only the stages after detailed design in the development process. These adjusted industry benchmarks were used, often referred to as physical function point measures, for projects with 600 function points in our comparison shown in Table 2. 600 function points were selected since the mean size of projects implemented in RSDC is 600. The sample of pilot projects included an equal number of mainframe and client-server projects and, hence, the average of mainframe and client-server benchmark numbers for our comparison was computed. The pilot teams produced double the number of function points per staff month from the previous company baseline and

more than double from the industry standard. Using the variance of the measures among the six pilot teams in an estimate of the variance of the means, the pilot team metrics are significantly different from the means of both the company baseline ($t = 5.67$, $p < .001$) and the industry standard ($t = 4.76$, $p < .001$). The pilot teams did not have a lower cycle time than either the company or industry standard. The team, sponsor, and end user satisfaction measures are shown in Table 3. These scales ranged from 1 = very dissatisfied to 5 = very satisfied. In summary, the pilot teams were remarkable in their productivity while not sacrificing team, sponsor, or end user satisfaction with the resulting products.

TABLE 4
Comparison of the Pilot with Subsequent Teams in the RSDC

	Pilot Teams	Subsequent Teams	*significance*
FP/Staff Month *(higher is better)*	29.49 (*SD* = 7.88)	49.28 (*SD* = 18.52)	*p<.01*
Cycle Time *(lower is better)*	7.64 (*SD* = 3.37)	6.34 (*SD* = 5.93)	*n.s.*

TABLE 5
Comparison of the Satisfaction of Pilot and Subsequent Teams in the RSDC

	Pilot Teams	Subsequent Teams	*significance*
Team Satisfaction	4.15 (*SD* = .25)	4.30 (*SD* = .40)	*n.s.*
Sponsor Satisfaction	4.56 (*SD* = .55)	4.29 (*SD* = .54)	*n.s.*
End User Satisfaction	3.68 (*SD* = .52)	3.97 (*SD* = .51)	*n.s.*

TABLE 6
Comparisons of Entry versus Exit Questionnaire Data

	Entry	Exit	*significance*
Preferences for war rooms	3.53 (*SD* = .36)	4.0 (*SD* = .46)	*p<.05*
Preferences for cubicles	3.86 (*SD* = .46)	3.43 (*SD* = .46)	*p<.05*

How the Facilities Impacted the Collaboration and Communication

The entry and exit questionnaires also revealed that attitudes about the activity in the room changed over time. As shown in Table 7, at project completion, individuals were significantly less distracted by the presence of others in the room ($t (5) = 3.64$, $p < .01$).

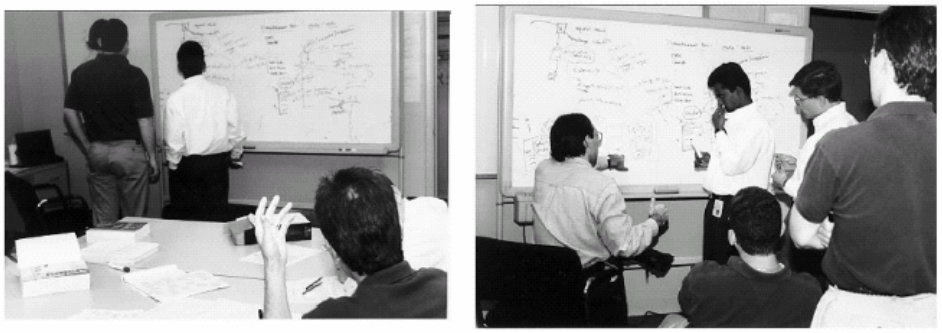


Fig. 2. Two phases of interaction in the RSDC: The team moving from two separate meetings to one central one, from overhearing.

TABLE 7
Changes in Reported Attitudes about Activity in the War Rooms

	Entry	Exit	*significance*
Susceptibility to distraction	3.37 (*SD* = .49)	2.68 (*SD* = .43)	*p*<.001
Concern about individual recognition	2.87 (*SD* = .31)	3.29 (*SD* = .49)	*p*<.025

IMPLICATIONS OF THE EXPERIMENT

The data from Teasley's study are striking. Groups working in the RSDC showed significantly improved productivity and high levels of satisfaction by everyone involved, from team member to customer. The significant improvements in productivity over the company baseline are most likely

TABLE 8
Nine Kinds of Work

1. Discussion to acquire customer input
2. Discussion of a political issue
3. Problem solving at the whiteboard
4. Status meeting using the to-do list (usually on a flip chart or the whiteboard)
5. Team building discussion (social)
6. Training
7. Simultaneous problem solving meetings (subsets of team members)
8. Working solo (typically coding)
9. Private conversations with outsiders

due to the tight fit between the development method (timeboxing) and the collaborative facilities. War rooms contributed to the enhanced software development because they constituted a collaborative information system, which facilitated communication and continual awareness [23]. There has been a follow-on to the pilot projects at this company. Because of the positive results for the pilot teams, the

company has built 112 war rooms in a facility solely dedicated to software development. Groups of eight war rooms constitute a "neighborhood" in which a central area houses the support services (e.g., the database expert, the methodologist, etc.). The rooms themselves are both large and configured in the "E" style used in a few of the pilot teams (see Fig. 1). The entire end wall of the room is whiteboard (and, indeed, they ask for stepladders so they can use the entire surface). In informal training or status meetings, teams project a laptop from a portable stand onto the whiteboard, which has a semigloss surface suitable for viewing. This paper has shown that, when people are radically collocated on a software development team, productivity goes up and timeliness increases. Collocation brings interactive, continuous communication, which allows overhearing and awareness of teammates' activities. This helps for clarification, problem solving, and learning. It also enhances team building.

FACTOR F_3. REDUCING REWORK COSTS

[IEEE Transactions On Software Engineering, Vol. 14, No. 10, October 1988]

One of the key insights in improving software productivity is that a large fraction of the effort on a software project is devoted to rework. This rework effort is needed either to compensate for inappropriately-defined requirements, or to fix errors in the specifications, code or documentation. For example, Reference [70] provides data indicating that the cost of rework is typically over 50 percent on very large projects.

A significant related insight is that the cost of fixing or reworking software is much smaller (by factors of 50 to 200) in the earlier phases of software life cycle than in the later phases [22], [44], [35]. This has put a high premium on early error detection and correction techniques for software requirements and design specification and verification such as the Software Requirements Engineering Methodology, or SREM [3], [4] and the Problem Statement Language/Problem Statement Analyzer [111]. More recently, it has focused attention on such rapid simulation [125], [109], which focuses on getting the right user requirements early and ensuring that their performance is supportable, thus eliminating a great deal of expensive downstream rework.

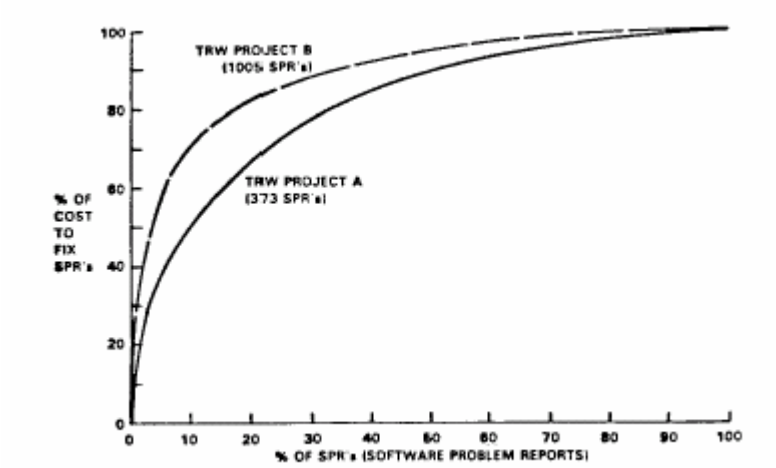


Fig. 4. Rework costs are concentrated in a few high-risk items.

Another important point is that rework instances tend to follow a Pareto distribution: 80 percent of the rework costs typically result from 20 percent of the problems. Graph above shows some typical distributions of this nature from recent TRW software projects; similar trends have been indicated in [102], [47], and [13]. The major implication of this

distribution is that software verification and validation activities should focus on identifying and eliminating the specific high-risk problems to be encountered by a software project, rather than spreading their available early-problem-elimination effort uniformly across trivial and sever problems. Even more strongly, this implies that a risk-driven approach to the software lifecycle such as the spiral model [27] is preferable to a more document-driven model such as the traditional waterfall model.

The natural solution to cut down rework cost in to reuse components. Intuitively, savings occur with software product reuse because reused components do not have to be built from scratch. Further, overall product quality improves if quality components are reused. With software process reuse, productivity increases to the extent that the reused processes are automated, and quality improves to the extent that quality-enhancing processes are systematized. Further, there is plenty of duplication in the applications being developed and maintained nowadays, and hence plenty of room for reuse. In 1984, for example, the U.S. software market offered some 500 accounting programs, 300 payrolls programs, 150 communication programs, 125 word-processing packages, etc. [77]; the figures are probably higher today. In the early eighties, Lanergan and Grasso estimated that 60% of business applications can be standardized and reused [85]. Generally, potential (estimated) and actual reuse rates range from 15% to 85% (see, e.g., [59], [103]). Existing experience reports suggest that indeed good – sometimes impressive – reuse rates, productivity and quality increases can be achieved (see, e.g., [12], [13], [73], [100]). However, successes have not been systematic (see e.g., [59], [133]), and a lot of work remains to be done both in terms of "institutionalizing" reuse practice in organizations and in terms of addressing the myriad of technical challenges that make re-use difficult [83].

TECHNICAL DETAILS OF REUSE

Adopting the transformational systems' view of software development as a sequence of transformations and/or translations of the description of the desired system from one language (level i description) to another (level i + 1 description) as shown in Fig. 1. Three levels of knowledge are used in this translation:

1) knowledge about the source domain (level i)
2) knowledge about the target domain (level i + 1), and
3) knowledge about how objects (entities, relations, structures) from the source domain map to objects in the target domain.

For a given level , the knowledge can be seen in linguistic terms, as consisting of a domain language, and a set of expressions known to be valid. The domain language consists of domain entities (or classes) and domain structures. The descriptions of the various entities and structures can be based on an enumeration of legal entities and structures, or based on a set of properties that must be satisfied by either (e.g., consistency checks, composition rules), or a mix of the two. We refer to the description methods as enumerated and compositional, respectively. The descriptions of past problem instances constitute the expressions that are known to be valid.

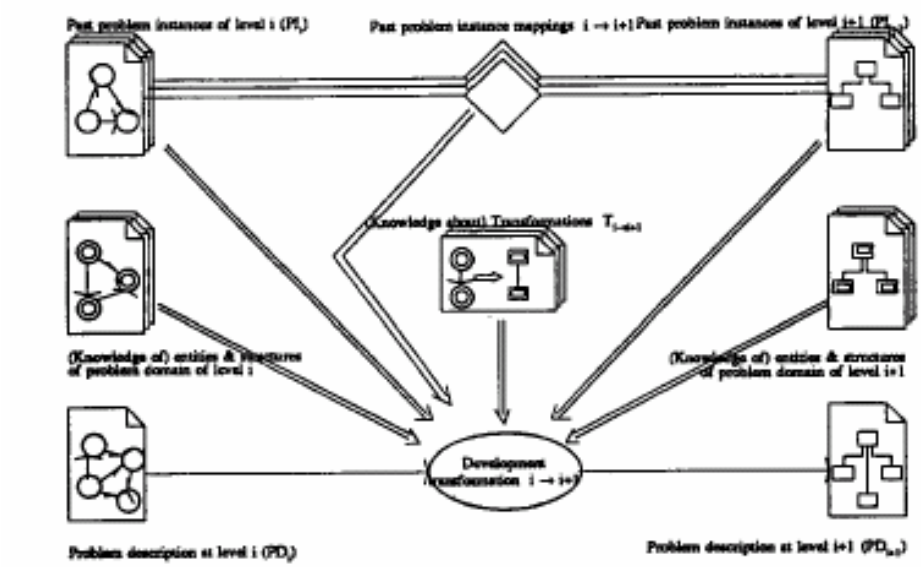


Figure 1. A categorization of reusable knowledge

The mapping knowledge consists of a set of transformation rules, from level I to level I + 1, and a set of known mappings between problem instances of level I and problem instances of level I + 1. The transformation rules embody what is usually referred to as process reuse or skill reuse (see, e.g., [133]). We shall refer to them as the transformational grammar. Note that this formalism does not distinguish between declarative knowledge and procedural knowledge as we feel the distinction to be a mainly representation issue.

Typically, development consists of, first, describing (specifying) the problem at hand in the language of level I to obtain a description PD_i and, second, transforming that description into one at level I + 1 (PD_{i+1}), supposed to be the target description language (e.g. executable code). With reuse, one would want to avoid having to manually,

1) Specify completely the problem at hand and/or
2) Transform the entire specification of level I into level I + 1

Thus, reusable assets include all the kinds of knowledge involved in the development transformation ($DT_{i->i+1}$), which can be thought of as the result of applying a generic level independent problem-solving method on the relevant knowledge sources. The various reuse approaches can be categorized based on:

1) The extent to which the language of level I covers the problem domain of level I and
2) The extent to which the mapping knowledge ($T_{i->i+1}$), covers all the entities and structures (i.e. all the valid expressions) of the domain of level i.

Finer characterizations may be based on the kind of language description used, along the enumerated versus compositional dimensions. Table II shows the characteristics of some of the approaches commonly referred to in the literature. As we go down the rows of Table II, we move from what is generally referred to as the ***building blocks*** approach to increasingly automated ***generative*** approaches. Automation requires the complete "cover" of the source domain language (level I) and the completeness of the mapping knowledge I -> i+1. In other words, automation is possible if we can express all new

problems in terms of problems, or combinations of problems, that have already been solved. We comment below on the various approaches separately.

With source code components, a new problem is solved by composing solutions to subproblems. A complete cover o level I domain would mean that all the components that one may need have been developed, or, more astutely – but equally unrealistic – a set of components that have been developed such that every problem can be reduced to subproblems that these components can solve. Notwithstanding the issue of finding such a decomposition/reduction, which can be as challenging as solving the original problem analytically from scratch, the number of required components is most probably prohibitive [83]. That number depends on:

1) the breadth of the application domain and
2) the composition technique used.

With source code components, composition often takes place "too late" in the software life cycle, limiting the range of behaviors that can be obtained from a set of components to variations on functional composition, as supported by traditional module interconnection languages (see, e.g., [129]) or programming languages. Source code components approaches that support composition of components at a higher level of abstraction yield a greater range of behaviors (see, e.g., [78], [149]). Software schemas are similar to source code components, except that the reusable artifacts are defined at a higher level of abstraction, allowing for a greater range of instantiations (through partial generation) and compositions. Further, the added parameterization makes it possible to build complex, yet generally useful structures (see, e.g., [16]). However, the artifacts are still not meant to cover all the needs of the application domain, and finding and expressing the right compositions are still challenging design problems.

With the remaining 3 approaches, the source domain language covers the application domain. Transformational systems fall short of automation because the mapping knowledge is incomplete or non-deterministic: A transformational system needs developer assistance in selecting among applicable – and perhaps objectively equivalent – transformations [123]. The transformational approach can be used in conjunction with source code components to assist in the modification and integration of such components in new applications [113]. Full automation is achieved with application generators and very high-level languages. With very high-level languages, automation is possible at the cost of code efficiency and design quality; very high-level languages are not intended to implement production quality software. Automation is possible with application generators because of a restriction of the application domain. The restriction has the added advantage of making it practical to enumerate a set of template software specifications (or the corresponding software "solutions") parameterized directly with user requirements.

It is fair to say that as we go down Table II, the focus shifts from components to composition, and the language for expressing compositions moves up in terms of abstraction. This corresponds closely to Simos's "reuse life cycle", which prescribes an evolution of reuse approaches within organizations, following the maturing of both the application domain and the expertise of developers within that domain [148].

TABLE II
A CATEGORIZATION OF COMMON REUSE APPROACHES

	Language Level I			Mapping Knowledge			
Approach	**Life Cycle Stage**	**Covering**	**Description Type**	**Covering**	**Description Type**	**Spectrum**	**Examples**
Source code components (see [50], [83])	mostly design	partial	compositional (mostly)	partial	compositional (mostly)	wide spectrum	RSL [27], REBOOT [116], and a number of other "nameless" tools and approaches (e.g. [85], [131], [161]). Object-orientation, seen as a development methodology for reusable components, is discussed in §IV.C. Problems related to the use of such components are discussed in various subsections §V.
Software schemas (see e.g. [83], or referred to as *reusable program patterns* in [77])	mostly design	partial	compositional (mostly)	partial (mostly)	compositional (mostly)	wide spectrum	The programmer's apprentice [137], the PARIS system [80], and Basset's *frame-based software engineering*, in which an application could be *completely* specified and generated using frames [16]. Software schemas are briefly discussed in the context of OO technology §IV.C).
Reusable transformation systems (see e.g. [19], [50])	Software specifications	complete	compositional	partial	compositional	wide-spectrum	A somewhat outdated survey of transformational systems is given in [123]; their potential for quality-preserving maintenance and reuse has been recognized by a number of researchers, including Feather [45], Arango et al. [4], and Baxter [17]. They are discussed in more detail in §V.C.
Application generators (see e.g. [83])	User requirements complete	complete	enumerated (mostly)	complete	narrow, domain-specific	narrow, domain-specific	Unix's Yacc, a number of commercial tools in business information processing (see e.g.[69] for a survey), a number of user interface building frameworks (see e.g. [119] for a survey), etc. Discussed in more detail in §V.B.
Very high-level languages [83], reusable processor [77], etc	Software specifications	complete	*Emphatically* compositional	complete	compositional	depends on the system	Simos' ASL are application-specific languages [148], PAISLey [162] SETL [82] and others are based on application-independent mathematical and computational abstractions. T}.TE.LP.ls 2.ce Table 2. A categorization of common reuse approaches..LP.sp.PP

COSTS AND BENEFITS OF REUSING SOFTWARE

The most vaunted advantages of software reuse are:

(1) An increase in the productivity of software development, which translates directly into monetary terms and
(2) An improvement of the quality of the products, which may mean less corrective maintenance, easier perfective maintenance, greater user satisfaction, and so forth, all of which translate into monetary gains

However, there are also different costs associated with software reuse, both capital setup (up-front) costs and proportional costs (cost-per-use). Further, different technical approaches to reuse have different investment and return on investment profiles (see, e.g. [42], [148]). Economic models and software metrics are needed that quantify the costs and benefits of reuse.

Three basic managerial decisions exist for software reuse:

(a) The decision to launch an organization-wide software reuse program (a long-term, capital investment-like decision [11], [52], [128])
(b) The decision to develop a reusable asset (a domain engineering decision [52] and

(c) The decision to (re)use a reusable asset in an application currently under development (an application engineering decision [52]).

To decide on the most suitable re-use decision, few factors have to be taken into account:

Reuse (White Box / Black Box) Cost

Reuse Cost is explained by Mili [1-S] in the context of a point in development where a developer has the option of building a component from scratch, but chooses instead to try to reuse a component from the library. 2 kinds of reuse exist: White Box Reuse and Black Box Reuse.

In Black Box Reuse, the component is integrated in its host environment without modifications, and the average cost of doing Black Box Reuse is given by:

$$[Search + (1-p) \times Development]$$

where search is the cost of performing a search operation on the database, *Development* is the cost of developing the component from scratch, and p is the probability that the component is found in the database. The reuse option is only attractive if:

$$[Search + (1-p) \times Development] < Development,$$
$$\text{or } Search < p \times Development.$$

To favor reuse, the library coverage has to be adequate (large p) to make sure that developers can quickly find the component they need or determine that it doesn't exist.

For White Box Reuse, the component is adapted and integrated into the host environment, and the developer must weight the cost of producing a component from scratch against the cost of attempting to reuse one, possibly after modifying it. The average cost of developing with intent to reuse can be formulated thus:

$$[Search + (1-p) \times (ApproxSearch + q \times Adaptation + (1-q) \times Development)]$$

where p is the probability that the component is found in the database, q is the probability that a satisfactory approximation of the component can be found, ApproxSearch is the cost of performing the approximate search, Search is the cost of performing an exact search operation on the database, Development is the cost of developing the component from scratch, and Adaptation is the cost of adapting the component to its host environment [11]. The reuse option is attractive if:

$$Search + (1-p)\ ApproxSearch + (1-p)\ q\ Adaptation \leq (p + (1-p)\ q)\ Development$$

If we consider that the fact that a satisfactory approximation of the component is found means that Adaptation is less than Development, then a sufficient condition for reuse to be attractive is given by:

$$Search + (1-p)\ ApproxSearch \leq p\ Development$$

which means the overall cost of search, whether a satisfactory component is found or not, is less than the savings that actually result from those (100p) % cases where a satisfactory component is found.

A developer who is well-versed with the contents of a component library can locate what he/she needs more quickly and knows when not to bother even looking. This has the effect of reducing the cost of individual searches (Search and ApproxSearch) and their relative frequency, which in case of perfect knowledge about the contents of the library, go down from *1* to *p* for exact search and from *1 - p* to *q* for approximate search.

REUSE PROGRAMS

It is a well-accepted fact that reuse increases software productivity, however, the question is not so much whether to set up a reuse program or not, but how. Reuse programs will inevitably vary across every organization, as there is no easy way to find out how much reuse is possible without actually doing it for a few years. To this end, Davis [42] proposed a set of guidelines (a reuse capability model) which helps organizations define the objectives of reuse in terms of 3 measures:

1) *reuse proficiency* – which is the ratio of the value of the actual reuse opportunities exploited to the value of potential reuse opportunities
2) *reuse efficiency* – which measures how much of the reuse opportunities targeted by the organization have actually been exploited and
3) *reuse effectiveness* – which is the ratio of reuse benefits to reuse costs

FACTOR F_4: CODE INSPECTION

BACKGROUND TO CODE INSPECTIONS

A software inspection is a formalized and rigorous review method. They were originally developed by IBM to improve software quality and increase programmer productivity. First described in writing by Fagan [1-1] in 1976, inspections have gradually gained popularity with companies other than IBM [1-2] and have even spread to other countries [1-3]. The aim of inspections is to decrease maintenance and product errors, in order to effect an overall increase in quality and productivity. Initially, the inspection method devised by IBM consisted of teams of three to six participants. This was then adapted by Bisant and Lyle, [1-S] to a leaner two-man variant of the original system as described below.

DESCRIPTION OF CODE INSPECTIONS

This technique involves pairing up coders in groups of 2 with the aim of code inspection. In this group, there will be a designer and an inspector in which the roles are switched at the end of the inspection process. The inspection process consists of the following steps:

1. Overview
2. Preparation
3. Inspection
4. Rework
5. Follow-Up

For Overview, the designer would describe the overall dimensions of the project and how his portion fits into that area. Preparation is the vehicle for individual education. The inspector studies the design, its intent, and its logic. The inspector should also study the ranked distributions of common error types, as well as clues on how to find common errors. During inspection, the inspector will describe how he will implement the design as it is expressed in the document. The objective during the inspection is to find errors. The errors are detected during the discussion. Any questions raised are pursued only to the point where the error is identified. The error is then noted and classified. If the solution is obvious, then it is noted otherwise the inspection continues. The inspection is not to redesign, consider alternative design strategies, or evaluate what is being inspected. A report on the inspection and its findings is then issued after a day. Rework is performed by the designer to correct all defects and follow-up is done by the inspector to ensure that all issues of concern have been covered and all rework has been completed.

All operations are necessary. Omitting or combining steps leads to inferior inspections outweighing any short term benefits. It is also important not to submit too much work to any one inspection session. The optimum duration of the inspection session is about 2 hours. 2 sessions per day are recommended and rework or follow-up must be scheduled so it is not postponed or avoided.

Part of the process of using inspections is the formation of a defect-classification scheme. This scheme must be defined for all inspection types used in an organization. Once the scheme is established, analysis of error data is straightforward and this information can be used for process control. Errors should be classified as to their type; whether they are missing, wrong, or extra; and whether they are major or minor. The defect specification is constructed by inspecting representative products and analyzing the errors found by type, origin, cause and salient indicative clues. Once the specification is constructed it can be used as an aid in teaching people how to find errors effectively and how to focus on high cost errors.

HOW INSPECTION IMPROVES PRODUCTIVITY

There are two main benefits of inspections. The first benefit of conducting code inspections is to make the resulting product more statistically predictable. This means that figures such as hours worked, lines of code and errors produced while coding fluctuate less. Predictable and reproducible data will allow management to use the statistics as a basis for estimating cost of future projects, or as a basis for conducting research. The second benefit of inspections is to reduce coding errors and to improve individual programming productivity. Several studies have been conducted to support these assertions.

STUDIES AND FINDINGS

With this inspection system, the next natural question to ask is how reliable this system is, and how predictable the results of inspections are. Holding all factors constant,

it was found that inspections generally reduced the occurrences of errors by 38% [1-S]. However, different teams of programmers with different experiences and abilities will produce inspections of different qualities. Buck (IBM) [1-8] conducted a significant study to determine the magnitude of this variability and the factors that contribute to this variation. His initial analysis demonstrated 2 populations:

1) Fast material coverage and low error detection
2) Slower material coverage and high error detection

However, he could not determine if the code being inspected by the second population was more error prone. In an experiment to address this question, Buck reviewed 106 inspections of a constant software module which was used for training purposes in IBM. Buck examined the variables: size of inspection team, rate of coverage, amount of preparation, and the number of major errors found. It was found that inspection coverage of code at the rate of 125 NCSS (Non Commentary Source-Code Statements) per hour or less found an average of 43 percent more major errors than inspections that proceed at a faster rate. Reports from inspections of variable quality design and code materials also showed that high quality inspections, as determined by proper rate of coverage, find more errors as Table 1 indicates.

Table 1
Inspection Rates* (NCSS/Hour)

Inspection Number	Recommended Preparation Rate	Actual Rate	Maximum Rate	Additional Errors Found When Rate is < Maximum
0	200	220	295	178%
1	100	100	135	112%
2	125	90	125	136%

* Refer to step 2 of the inspection process
Source: Fagan [1-8]

Given the rate of coverage, team size (3, 4 or 5 members including moderator) does not significantly affect error detection. Large team sizes were found to have decreases in individual preparation time. It was also found that more preparation increased the likelihood of holding the inspection at the planning rate coverage. However, preparation could not compensate for improper coverage. One other finding of this study indicated the larger the amount of material to be inspected, the lower the percentage of recommended preparation time was put in by the inspection teams. As was briefly mentioned above, analysis was also performed on 520 variable quality code inspections. The analysis of these reports came to the same conclusions concerning team sizes and rates of coverage. Once again, about the only factor which was not examined in this study was the initial level of expertise of the participants.

As can be deduced from the research just presented, the formal and rigorous structure of inspections makes the data gathered during them much less variable. Since the results are repeatable and of high scientific quality, confidence can be placed in the data for use in process control. Inspections overcome the difficulties of assuring consistently reproducible data under conditions of diverse staffing and products. Management can use the data from inspections in several ways. The data can be used as feedback to improve the software development process and the inspection process itself.

Another experiment conducted by Bisant and Lyle[1-S], attempted to test a formal method for software inspections which used a leaner 2-person team. This was done by eliminating the role of the moderator. This modification was created with small organizations (in which forming large teams is impossible) in mind. Also, the experiment strived to verify the implication in Buck's study that 2-man teams are just as effective as larger teams. Verification of this fact would allow management to save significant manpower. Lastly, Bisant and Lyle's study aimed to shed light on the dynamics of the team error-detection process.

The experiment was conducted at the University of Maryland-Balitmore County, using third year undergraduate students in Computer Science as the subjects of the study. Research by Soloway and Erlich [1-11] has revealed that students at this level have developed similar cognitive organizational structures and strategies to those used by experts. Rudimentary development of these structures and strategies were even seen after only 3 programming courses. So, although undergraduates at this level are not the same as experts, they have taken significant steps in that direction. Additionally, the tasks and measurements that would be required fit well into the present data structures curriculum. Although high experimental mortality is expected with undergraduates, the experimental design was able to partially compensate for this. Since most groups of programmers are not homogenous with respect to ability despite similar background, assuming homogeneity can bias results. Despite this, most studies proceed under this assumption. This experiment used a design which is used rarely in the computer software engineering field although it most likely is used in other fields that are empirically oriented. This design is the Pretest-Posttest Control Group design as outlined by Conte, Dunsmore, and Shen [12]. It is relatively unaffected by lack of homogeneity as this can be compensated by modeling in the analysis. Using a control and an experimental group, a program was given to all subjects and a productivity measurement taken for that program.

Each class member was given two programming assignments which could be coded in the high-level language of his or her choice as long as dynamic memory was supported in that language. Each member had to use the same system and language on both programming assignments. The first program consisted of constructing a binary search tree and printing it out sideways. The students were given two weeks to complete the assignment and they were also provided with the basic algorithms necessary. The programming assignment also included a design to be written using a basic pseudocode which was provided. About 20 minutes was spent teaching them how to design and how to use the pseudocode. They were not graded on the design for the first assignment; instead, they were given feedback and graded slightly on the design for the second programming assignment. Designs for both assignments were to be at the level of detail where one design statement equaled between 3 and 5 source statements. Along with feedback for the first design, the students were given a completed design as an example of what was expected. Each student was also asked to keep a time sheet and record to the nearest 15 minutes the amount of time spent in the morning, afternoon, and evening of each day. They were also asked to record time for any activity relating to the project such as studying, reading, coding, etc. During the entire experiment, the students were reminded at the start of each class and given a few minutes to update their time sheets if they had not already done so.

Up to the second programming assignment, both the experimental and control groups did the exact same thing.

For the second programming assignment, however, the experimental group was asked to perform an inspection along with a classmate of each other's design or of each other's code, or of both. For each inspection, 10 minutes was spent instructing the students in the proper technique along with an introduction to the types of errors to look for and how to find them. The subjects were told not to rush and not to worry if they did not complete the inspection in the time given. The students were required to bring two copies of whatever product was to be inspected. They then paired off, spent 20 minutes inspecting one of the products, switched, and spent another 20 minutes on the other. At this point, the subjects took a few minutes to hand in the materials and update their time

The design inspection instruction was more detailed than that for the code inspection as much of the information was the same for both types of inspections and did not need repeating. A brief mention was made why inspections are performed. No overview or preparation was required of the students as they were all given the same assignment and group education was not necessary. The students, however, were given 5 minutes to look over the partner's design before the inspection started. The subjects were told the partner who did not write the design should read or describe how he will implement the design. They were told to read through the design, paraphrasing as they went. Every piece of logic was to be covered at least once and every branch was to be taken at least once. The objective of finding errors was stressed heavily. An error was defined as the following:

- An instance of insufficient detail

- A parameter or routine not defined or defined incorrectly

- A portion of design which, if implemented as stated, could cause a malfunction or error in the program.

The subjects were told to look for errors as they progressed through the design. Any questions were to be pursued only to the point where the error was recognized. The error was to be identified and if the solution was obvious, it was to be noted; otherwise the inspectors were to continue through the design. They were told not to redesign, not to evaluate alternative design solutions, and not to try and find solutions to errors. Several examples of errors they might find were provided. The subjects were told to note the errors on a sheet of paper and turn it in after the inspection along with one copy of the product. Follow-up was implemented by requiring the students to hand in a corrected design at the next class meeting.

The code inspection was implemented 4 days after the design inspection and 3 days before the program was due. The students were asked to bring two copies of a compiled listing, and two copies of the corrected design. The subjects were told the partner who did not do the coding was to read through the code making sure it implemented the design. Every piece of logic and every branch were to be covered at least once. Again the objective of finding errors was stressed and the subjects were instructed not to try and find solutions and not to evaluate alternative design solutions. Examples of errors were also given to the subjects and the errors found during the inspection were to be noted and handed in along with the compiled listing. There really was not a follow-up for this inspection as it was not necessary since the program was due three days later.

Approximately 55 students began the experiment and 32 completed it. Quite a few students had already dropped the class from their schedules when the experiment began and some of this continued as the experiment progressed. This reduction in class size occurs frequently at the undergraduate Level and it was the most significant cause of the experimental mortality in this study. Students were also monitored very closely to ensure plagiarizing did not take. A few students were removed from consideration for this reason.

This high experimental mortality was not expected to bias results as the mortality comes from the poorer programmers. This type of programmer is essentially removed from consideration across all sections equally. It would be a concern if the poorer programmers in one group were dropped while the better programmers in the other group were dropped. That is usually not the case and it certainly was not in this study.

Many deliverables along with the tasks were required for a student to be considered a member of the study. The requirements for each member of the control group were:

- a design, completed program, and time sheet for Part 1
- a design, completed program, and time sheet for Part 2

The requirements for each member of the experimental group were the same plus one additional requirement:

- an inspection of their design or an inspection of their code or both for Part 2, error reporting sheets and products were handed in.

The important requirements for this type of design are that the programs are similar and that each individual subject uses the same language and system for both programs. With these requirements met, it can be assumed that the programming rates for both programs are similar. Each member of the experimental group selected VAX Pascal as the choice of language. This facilitated the inspecting of each others code as each member would have knowledge of the coding language being inspected.

Although this study was designed primarily to measure productivity, data was also collected on the number and types of defects found for each inspection. Unfortunately, this data could not be analyzed due to the variable specificity with which it was reported. Although students were given some examples on how they might report errors, these examples were not followed by all students. A standard reporting form might have been better. A conclusive analysis of those students performing design inspections versus those performing code inspections versus those performing both could not be done due to the small sample sizes that would be involved.

The results indicated there was a statistically significant improvement in productivity of the experimental group over the control group. The amount of improvement appeared to increase for the poorer programmers.

RESULTS

The experimental procedures produced a set of measures for each subject. These measures included the time to complete the first program (Time1). the time to complete the second (Time2). the age (Ages), the experience in months (Experien). the

number of courses (Courses), the number of courses in the language they were using (Pascal, named this since most of the subjects used Pascal). the number of different languages they knew (Langs), and whether their status was professional or student (Stat).

The analysis chosen was a one-way analysis of variance with a continuous covariate. The covariate was the Time I pretest measurement. A model was constructed to express the relationship between the covariate (Time 1) and the response variable (Time2). A separate model was constructed for the experimental and the control group. The analysts then examined the two different models to see if there was a significant difference between the two.

Complete mathematical details of Bisant and Lyle's study can be found in the appendix.

The final model appeared to be:

$$\textbf{LTime2(i, LTime1)} \ \alpha_i + \beta_i \textbf{LTimel.}$$

The actual parameter estimates were:

Control	**LTime2 = 0.49 + 0.91 LTime1**
Experimental	**LTime2 = 1.95 + 0.298LTime1**

By taking the antilogarithm of both sides the power equations below are obtained:

Control	$\textbf{Time2 = I.63(Timel)}^{0.91}$
Experimental	$\textbf{Time2 = 7.03(Time1)}^{0.298}$

The difference between these two models was apparent when some actual values were substituted for the Time 1 variable. The average value for Time 1 for both groups was 15.54. When this value was substituted into the control group model, we obtained a value of 19.79 for Time2. When 15.54 was substituted into the experimental group model, a value of 15.92 was obtained for Time2.

These results indicated there was a significant improvement in the experimental group as a result of using the two-person inspection method. The slower the programmers, the more improvement was observed. No comparison could be made for the very fast programmers.

RESEARCH CONCLUSIONS

The two-person inspection method appeared to improve novice programmer productivity. This conclusion is formed from the significant difference found between the power equations used to model both groups. This improvement probably came about by detecting and removing errors early in the development process when it required the least amount of time. The two-person inspection process probably imparted problem knowledge to the participants faster than those who did not use the inspections and this might have accounted for some of the improvement. This is in agreement with previous research (Buck) on larger team inspections. It appeared as though the less productive, programmers may have benefited the most from the technique. The improvement observed in this experiment would most likely generalize to professional programmers although possibly not to the same degree.

Some of the added benefits of inspections include the detailed feedback given to programmers on a real-time basis. The programmer learns what types of errors he is prone to make and he can concentrate on avoiding these errors and make appropriate corrections to his programming style so that his performance improves on future units. The results of inspections should never be used as an appraisal of performance since a programmer might consciously or subconsciously inhibit the inspection process to make himself look better. An additional benefit is the high degree of product knowledge imparted to the inspection participants in a short amount of time. This improves the ability of the participants in later development and testing.

For the manager concerned with Software productivity, the most important implication of this inspection method was an improvement in individual productivity. This could be helpful to individual programmers or those programmers that do not have access to larger team resources. It should be noted for our purposes that the unit of measurement for the study just described is somewhat unusual. It does not use Lines of Code anywhere in the analysis; rather, the focus is on the functionality and correctness (i.e. it satisfies the specified criteria given) of the code produced and the time taken to produce it. Also, less experienced programmers tend to benefit more from this scheme, thus it would be appropriate for the manager to make proper adjustments for this.

3 Other Factors

Other than the 4 factors mentioned above, there are numerous other factors which can also affect Software Productivity. Many of these factors have been studied by researchers and some of these studies are listed in the table below, along with the number of projects researched and the measure of productivity used in the research.

TABLE 1
OVERVIEW OF SOME PRODUCTIVITY FACTORS CONSIDERED IN PAST RESEARCH

Some Major Productivity Factors	ESA Data	A*	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
Country	X											X			X				
Company	X														X		X		
Category/Type	X											X	X		X		X		
Industrial/Business Environment	X												X		X				
Language	X	X	X			X							X	X					
Team Size	X						X			X	X								
Duration	X	X					X									X	X		
Project Size	X		X			X			X		X		X				X	X	
Required Software Reliability	X							X											
Execution Time Constraint	X							X	X									X	X
Main Storage Constraint	X							X	X									X	X
Virtual Machine Volatility	X							X											
Programming Language Experience	X			X				X	X					X					X
Modern Programming Practices	X		X	X	X			X	X	X			X	X	X	X		X	X
Tool Use	X				X			X		X			X	X		X			
Product Complexity				X				X	X					X	X	X		X	X
Analyst Capability					X			X						X		X			
Applications Experience				X	X			X	X					X		X			X
Programmer Capability				X	X			X						X	X				
Virtual Machine Experience				X		X		X	X					X					X
Amount of Documentation							X			X									X
Overall Personnel Experience										X			X	X				X	X
Customer Interface Complexity				X					X					X				X	X
Design Volatility				X					X	X				X		X		X	X
Hardware Concurrent Development									X									X	X
Quality Assurance					X					X						X			
Development Environment (Online)			X			X								X	X				

* *Table 2 describes the research referred to by each letter.*

TABLE 2
OVERVIEW OF MAJOR DATABASES WHICH INCLUDE PRODUCTIVITY FACTORS

Database Code in Table 1	Reference	No. Projects	Environment	Geographical Scope	Productivity Measure
ESA Data	Maxwell 1996	99	Space/Military/Industrial	Europe	L.O.C.
A	Aron 1976	9	IBM Large Systems	USA	L.O.C.
B	Albrecht 1979	22	IBM Data Processing	USA	F.P.
C	Bailey and Basili 1981	18	Space (NASA/Goddard)	USA	L.O.C.
D	Banker et al. 1991	65	Commercial Maintenance Projects	USA	F.P. and L.O.C.
E	Behrens 1983	24	Data Processing	USA	F.P.
F	Belady and Lehman 1979	37	Not Identified	USA	L.O.C.
G	Boehm 1981	63	Mixture	USA	L.O.C.
H	Brooks 1981	51	Mixture (from Walston-Felix)	USA	L.O.C.
I	Card et al. 1987	22	Space (NASA/Goddard)	USA	L.O.C.
J	Conte et al. 1986	187	Mixture	USA	L.O.C.
K	Cusumano and Kemerer 1990	40	Mixture	USA and Japan	L.O.C.
L	Jones 1991	4,000	Mixture (primarily Systems and MIS)	USA	F.P. (converted from L.O.C.)
M	Kitchenham 1992	108	Not Identified (probably Commercial)	Europe (1 company)	F.P.
N	Lawrence 1981	278	Commercial	Australia	L.O.C.
O	Nevalainen and Mäki 1994	120	Commercial	Finland	F.P.
P	Putnam and Myers 1992	1,486	Mixture (primarily Business Systems)	Primarily USA, also Canada, Western Europe, Japan, and Australia	L.O.C.
Q	Vosburgh et al. 1984	44	Mixture (from ITT)	9 countries	L.O.C.
R	Walston and Felix 1977	60	Mixture (from IBM)	USA	L.O.C.

Source: IEEE Transactions on Software Engineering, Vol 22, No. 10, Oct 1996

MEASURES OF PRODUCTIVITY

Management Guru Peter Drucker once said "That which you cannot measure, you cannot manage". How then can we measure something as abstract as Productivity, and furthermore, how do we utilize the figures obtained from our equation and put them to use?

As can be seen, there has been many different measures of productivity and throughout the studies detailed in this paper, the measures of productivity has been different. The most convenient and widespread measure of productivity is LOC (Lines-of-code Productivity) because this is easily measurable. The equation for LOC Productivity is given by:

$$\text{LOC Productivity} = \frac{\text{SLOC}}{\text{Man Months of Effort}}$$

A more recent (and complicated) variant of LOC productivity is process productivity as developed by Putnam and Myers [32].

This measure is defined as:

$$\text{Process Productivity} = \frac{\text{SLOC}}{\left(\frac{\text{Effort}}{B}\right)^{1/3}\left(\text{Time}\right)^{4/3}}$$

Where SLOC is developed, delivered lines of source code, Effort is the manpower applied to the work measured in manmonths or manyears, B is a skills factor which is a function of system size, and Time represents the duration of the work measured in months or years.

Of course, the LOC measure has its limitations, because although lines of code is a measure of output, this metric is suitable only when comparing projects developed in the same language. Lines of code are not comparable across different languages; a line of code in COBOL is not equivalent to a line of code in Java. Similarly, other more complicated measures such as object point metrics are more appropriate when the projects being compared all use object-oriented design.

The other most popular measure is Function point (FP) Analysis. This measure is an abstract but workable surrogate measure for the output produced by software projects that does apply to heterogeneous projects. This metric is appropriate for measuring productivity and the performance of application software projects and is widely used in software organizations [25], [3]. Function points are the weighted sum of five different factors related to the application's functionality. These factors are: Inputs, Outputs, Logical files, Queries, and Interfaces. The IFPUG (International Function Points User Group) method of function point analysis is to first compute a raw function point score based on a weighted count of the number of the five factors in the application. This raw score is then adjusted to control the inherent complexity in the projects based on inputs collected on 14 complexity factors. These factors range from complexities in the use of data communication features, transaction rate, and data volume, differences in performance objectives, online data updating features to complexities from multiple sites, and reusability of the application. In practice, companies usually hire IFPUG certified function point experts with years of experience in the field to do the analysis.

4 Productivity Function

SOFTWARE PRODUCTIVITY FUNCTION

It would be helpful to the manager to be able to assess the productivity level of his firm. Indeed, given a mathematical function of how various managerial factors affect productivity, the manager can then determine an organizational strategy for increasing productivity, while staying within budget.

While it is unrealistic to devise a universal measure of productivity across all firms, there are some common steps involved in devising a productivity function for the firm. First off, the manager can decide on a suitable measure of productivity (LOC/FP etc) based on the goals of the organization. Then, he can decide which factors of productivity are relevant in his firm. For example, colocating the entire project team may be feasible for small software companies but may just be impractical for software giants like Microsoft.

With the different factors of productivity, the manager can construct a software productivity function, letting the measure of productivity be the dependent variable, while making each factor listed above the explanatory variables.

A regression function:

$$Y_i = a_1 + a_2X_2 + a_3X_3 + a_4X_4 + \ldots + a_nX_n + u_i$$

(where Y is the productivity and X are the explanatory variables). Can be constructed and the coefficients of the regression can be determined by running regression tests. With this productivity function and the coefficients, managers can determine the extent of which the factors affect productivity.

Of course, as with normal regression, problems such as multicollinearity, heteroskedasticity, autocorrelation and measurement may be present. For each of these problems, we may apply the standard techniques to overcome them. For example, team size and collocation are highly correlated (multicollinearity) because they both have to deal with communication in the organization. In this case, it is difficult or impossible to isolate their individual effects on the dependent variable. But multicollinearity can some times be overcome or reduced by collecting more data, by utilizing a priory information, by transforming the functional relationship, or by dropping one of the highly collinear variables.

Another plausible problem is heteroskedasticity: If the OLS assumption that the variance of the error term is constant for all values of the independent variables does not hold, we face the problem of heteroskedasticity. This leads to unbiased but inefficient (ie, larger than minimum variance) estimates of the standard errors (and thus, incorrect statistical tests confidence intervals). One test for heteroskedasticity involves arranging the data from small to large values of the independent variable, X, and running two regressions, one for small values of X and one for large values, omitting, say, one-fifth of

the middle observations. Then, we test that the ratio of the error sum of squares (ESS) of the second to the first regression is significantly different from zero, using the F table with (n-d-2k)/2 d.f, where n is the total number of observations, d is the numbers of omitted observations and K is the number of estimated parameters.
If the error variance is proportional to X2 (often the case), heteroskedasticity can be overcome by dividing every term of the model by X and then re-estimating the regression using the transformed variables.

Other than statistical errors, the manager must face the problems of data collection and verifying that the data being used is accurate. But most importantly, the manager has to tailor the software productivity regression model to fit the needs of the organization, and to design suitable ways to test the model.

CONCLUSION

There are many factors that affect Software Productivity in an organization. The extent to which these factors influence productivity varies across firms and depends on the type of project. Key factors that can improve productivity to a great extent are improving communications, and to re-use as much software components as possible.

It will be helpful to the manager if there was a systematic way of approaching the factors that increase productivity, so as to find the most cost effective way of increasing output. To this end, it is recommended that the manager first defines a measure of productivity and then construct a regression function to determine the extent of which each factor affects this measure.

With this approach, the manager will be able to gauge the effectiveness of the factors of productivity, and with this understanding, make managerial decisions to improve productivity in the organization.

5 Appendix

MATHEMATICAL DETAILS FOR THE BISANT AND LYLE'S STUDY

Table II summarizes the Timel and Time2 measures. A logarithmic transform was used on the Time 1 and Time2 measurements to reduce the heterogeneity of the residual variance and to allow for the use of linear regression in the analysis. In taking the natural logarithm of Timel and Time2, LTime1 and LTime2 were produced. A scatter diagram of LTimel versus LTime2 is shown in Fig I with the regression lines for both groups shown.

The regression line with the greater slope is the line for the control group.

TABLE II
SIZES, MEANS, AND STANDARD DEVIATIONS OF TIME1 AND TIME2

GROUP	N	STATISTIC	Time1	Time2
exp	13	mean	17.96	16.85
		sd	7.85	4.89
con	16	mean	13.58	18.25
		sd	6.39	10.01
both	29	mean	15.54	17.82
		sd	7.29	8.02

The regression lines indicated a difference between the two groups. To see if the difference was significant, a model was constructed as shown below,

$LTime2(i, LTimel) = \alpha_i + \beta_i LTime1$

where i represents either the experimental group or the control group. In other words, The model represented a linear relationship between LTime2 and LTime1, For each group, there was a separate slope (β) and intercept (a) describing the linear relationship for that group. Computer analysis revealed a coefficient of determination (R^2 of 0.58 meaning the model explained 58 percent of the variation observed in LTime2. Several F tests showed the model was significant (p = 0.0001), there was a shift in location between the two groups once correction was made for the covariate LTime1 (p = 0.0277), at least one of the slopes was not zero (p = 0.0001), and the slopes of the two lines were different (p = 0.0068). The F tests above assume that the residuals (difference between the actual and expected values) of the model are normally distributed. To validate this assumption, a Shapiro-Wilk W test was made on the normality of the residuals with the null hypothesis "the data is normally distributed." The hypothesis was not rejected for the control group (p = 0.128), the experimental group (p = 0.4). or for both groups together(p = 0.128).

The appropriateness of the model was evaluated by examining the scatter diagrams of the residuals versus the other variables for which data had been collected. Two of the typical scatter diagrams are shown for the LTime1 variable in Fig. 2 and for the Experien

variable in Fig. 3. it was clear the residuals were distributed with the same variability over the range of the variables indicating there were no effects from these variables which were not al ready explained by the model. Similar results were ob served for the other variables (Age. Courses, Langs, and Stat). An exception was the Pascal variable representing the number of courses in the language the subjects were using. The scatter diagram for this variable is seen In Fig. 4. When added to the model, it raised the coefficient of determination to 0.68. However, the model was less significant ($p = 0.001$ 1), there were no longer significant shifts in location ($p = 0.1483$ for the group effect, $p = 0.2918$ for the Pascal effect), and the difference in slopes was less significant ($p = 0.043$). More importantly, however, the effect of the Pascal variable was counterintuitive. The observed effect indicated the more courses the subjects had in the language they were working with, the longer it took them to complete the program. This inconsistency along with the small effect of the Pascal variable caused suspicion that a clustering took place due to the limited representation for the different levels of that variable. For this reason, the variable was not included in the model.

6 Sources

[1] M. Alavi, "An Assessment of the Prototyping Approach to Information Systems Development," Comm. ACM, vol. 27, no. 66, 1984.
[2] M. Alavi, R. Nelson, and R. Weiss, "Strategies for End-User Computing: An Integrative Framework," J. MIS, vol. 4, no. 3, 1988.
[3] A. Albrecht and J. Gaffney, "Software Function, Source Lines of code, and Development Effort Prediction: A Software Science Validation," IEEE Trans. Software Eng., vol. 9, no. 6, pp. 639-647, Nov. 1983.
[4] T.J. Allen, Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization. Cambridge, Mass.: MIT Press, 1977.
[5] R.D. Banker, S. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Software Maintenance Costs," Comm. ACM, vol. 36, pp. 81-93, Nov. 1993.
[6] D. Barstow, "Artificial Intelligence and Software Engineering," Proc. Ninth Int'l Conf. Software Eng., vol. 9, no. 5, pp. 541-561, 1987.
[7] R.L. Baskerville and J. Stage, "Controlling Prototype Development through Risk Analysis," MIS Quarterly, vol. 20, no. 4, 1996.
[8] F. Becker and F. Steele, Workplace by Design: Mapping the High Performance Workscape. San Francisco: Jossey-Bass, 1995.
[9] B.W. Boehm, Software Engineering Economics. New York: Prentice Hall, 1981.
[10] C.V. Bullen and J.L. Bennett, "Groupware in Practice: An Interpretation of Work Experiences," Computerization and Controversy, second ed., R. Kling ed., pp. 348-382, 1992.
[11] H.A. Clark, Using Language. Cambridge, U.K.: Cambridge Univ. Press, 1996.
[12] L.M. Covi, J.S. Olson, E. Rocco, W.J. Miller, and P. Allie, "A Room of Your Own: What Do We Learn About Support of Teamwork from Assessing Teams in Dedicated Project Rooms," Cooperative Buildings: Integrating Information, Organization and Architecture: Proc. First Int'l Workshop, 1998.
[13] B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," Comm. ACM, vol. 31, no. 11, pp. 1268-1287, 1988.
[14] Cognition and Communication at Work, Y. Engestrom and D. Middleton, eds. Cambridge, U.K.: Cambridge Univ. Press, 1996.
[15] N.E. Fenton, Software Metrics: A Rigorous Approach. New York: Chapman and Hall, 1996.
[16] M.D. Fraser, K. Kumar, and V.K. Vaishnavi, "Strategies for Incorporating Formal Specifications in Software Development," Comm. ACM, vol. 37, no. 10, 1994.
[17] W.W. Gibbs, "Software's Chronic Crisis," Scientific Am., pp. 86-95, Sept. 1994.
[18] C.M. Giglio, "Business Is War, But You Can Fight Back with a Killer APP–A Virtual War Room" Infoworld, vol. 21, no. 18, p. 72, 1999.
[19] J. Greeno and J. Moore, "Situativity and Symbols: Response to Vera and Simon," Cognitive Science, vol. 17, no. 1, pp. 49-59, 1993.
[20] J.D. Herbsleb, A. Mockus, T.A. Finholt, and R.E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed," Proc. 23rd Int'l Conf. Software Eng., pp. 81-90, May 2001.
[21] D.J. Hoch, C.R. Roeding, G. Purkert, and S.K. Lindner, Secrets of Software Success. Boston: Harvard Business School Press, 2000.

[22] E. Hutchins, Cognition in the Wild. Cambridge, Mass.: MIT Press, 1995.
[23] E. Hutchins, "Constructing Meaning from Space Gesture and Speech," Discourse, Tools, and Reasoning: Essays on Situated Cognition, L.B. Resnick, R. Saljo, C. Pontecorvo, and B. Burge, eds., pp. 23-40, 1997.
[24] J. Johnson, "CHAOS: The Dollar Drain of IT Project Failures," Application Development Trends, vol. 20, no. 1, pp. 41-44, 1995.
[25] C. Jones, Applied Software Measurement: Assuring Productivity and Quality. New York: McGraw-Hill, 1996.
[26] C. Jones, Software Assessments Benchmarks and Best Practices, New York: Addison-Wesley, 2000.
[27] T. Kidder, The Soul of a New Machine. New York: Avon Books, 1982.
[28] R. Kraut, C. Egido, and J. Galegher, "Patterns of Contact and Communication in Scientific and Research Collaboration," Intellectual Teamwork: Social and Technological Foundations of Collaborative Work, J. Galegher, R.E. Kraut, and C. Egido, eds., pp. 149-171, Hillsdale, N.J.: Erlbaum, 1990.
[29] R.E. Kraut and L.A. Streeter, "Coordination in Large Scale Software Development," Comm. ACM, vol. 38, no. 7, pp. 69-81, 1995.
[30] M.S. Krishnan and M.I. Kellner, "Measuring Process Consistency: Implications for Reducing Software Defects," IEEE Trans. Software Eng., vol. 25, no. 6, pp. 800-816, Nov./Dec. 1999.
[31] M.S. Krishnan, S. Kekre, C.H. Kriebel, and T. Mukhopadhyay, "An Empirical Analysis of Productivity and Quality in Software Products," Management Science, vol. 46, no. 6, pp. 745-759, June 2000.
[32] K.C. Laudon and J.P. Laudon, Management Information Systems: New Approaches to Organization and Technology: New Approaches to Organizations and Technology. New York: Prentice Hall, 1998.
[33] G.M. Olson and J.S. Olson, "Distance Matters," Human Computer Interaction, vol. 15, pp. 139-179, 2001.
[34] J.S. Olson and S.D. Teasley, "Groupware in the Wild: Lessons Learned from a Year of Virtual Collocation," Proc. Conf. Computer Supported Cooperative Work, pp. 419-427, Nov. 1996.
[35] J.C. Nunnally and I.H. Bernstein, Psychometric Theory. New York: McGraw-Hill, 1994.
[36] D.E. Perry, N.A. Staudenmayer, and L.G. Votta., "People, Organizations, and Process Improvement," IEEE Trans. Software, vol. 20, no. 7, pp. 36-45, July 1994.
[37] S.E. Poltrock and G. Engelbeck, "Requirements for a Virtual Collocation Environment," Information and Software Technology, vol. 41, no. 6, pp. 331-339, 1999.
[38] Perspectives on Socially Shared Cognition. L.B. Resnick, J.M. Levine, and S.D. Teasley, eds. Washington: APA Press, 1991.
[39] B.R. Rich and L. Janos, Skunk Works. Boston: Little, Brown and Company, 1994.
[40] E. Rocco, T.A. Finholt, E.C. Hofer, and J.D. Herbsleb, "Out of Sight, Short of Trust," Presentation at the Founding Conf. European Academy of Management, Apr. 2001.
[41] W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," WESCON Western Electronic Show and Convention, 1970.
[42] W. Royce, Software Project Management: A Unified Framework. Addison-Wesley, 1998.
[43] S. Sawyer, J. Farber, and R. Spillers, "Supporting the Social Processes of Software Development Teams," Information Technology and People, vol. 10, no. 7, pp. 46-62, 1997.
[44] S.D. Teasley, L. Covi, M.S. Krishnan, and J.S. Olson, "How Does Radical Collocation Help a Team Succeed?" Proc. ACM Conf. Computer Supported Cooperative Work (CSCW `00), pp. 339-346, Dec. 2000.

[45] S.D. Teasley and J. Roschelle, "Constructing a Joint Problem Space: The Computer as a Tool for Sharing Knowledge," Computers as Cognitive Tools, S.P. Lajoie and S.D. Derry eds., pp. 229-258, 1993.
[46] Java Technologies: Case Studies, Edward Jones: Retirement Planning System: http://www.sun.com/java/javameansbusiness/edward jones, 1999.
[47] "How to Jump-Start Java Computing Initiatives," Sun J., Nov. 1999, http://www.sun.com/SunJournal/v1n4/global2.html.
[48]*Sofware Engineering Journal, November 1991*
[49] *IEEE Transactions On Software Engineering, Vol. 28, No. 7, July 2002*
[50]Rapid software development through team collocation
Teasley, S.D.; Covi, L.A.; Krishnan, M.S.; Olson, J.S.;
[51] Software Engineering, IEEE Transactions on , Volume: 28 Issue: 7 , Jul 2002
Page(s): 671 -683
[52] *IEEE Transactions on Software Engineering, Vol 15, No 10 Oct 1989]*
[53] M E. Fagan, "Design and code inspections to reduce errors in pro gram development IBM Sysr. J., vol 15. no. 3. pp. 182-211, 1976