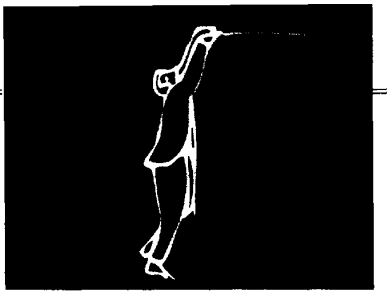


People, Organizations, and Process Improvement

DEWAYNE E. PERRY, AT&T Bell Laboratories
NANCY A. STAUDENMAYER, MIT Sloan School of Management
LAWRENCE G. VOTTA, AT&T Bell Laboratories

◆ *In their efforts to determine how technology affects process, researchers often overlook organizational and social issues. The authors report on two experiments to discover how developers spend their time. They describe how noncoding activities can use up development time and how even a reluctance to use e-mail can influence the development process.*

To understand the processes by which we build large software systems, we must consider the larger development picture, which encompasses organizational and social as well as technological factors. The software community pays too much attention to the technological aspects of software development at the expense of these other contexts. One often-cited reason is the difficulty of quantitatively measuring people factors. We don't see this as a valid argument. What is often assumed to be qualitative, such as social interactions, can often be quantified. We also believe that a holistic measurement-based approach that encompasses all the relevant contexts — both technological *and* social and organizational — is a prerequisite to genuinely understanding development processes. Without that understanding we cannot hope to significantly improve these processes and justify their improvement.

In this article, we describe two experiments that are the first in a series to enhance our understanding of the structure of software-development processes, with the goal of reducing the process-development interval. The first experiment was to see how programmers thought they spent their time by having them fill out a modified time card reporting their activities, which we called a time diary. In the second experiment, we used direct observation to calibrate and validate the use of time diaries, which helped us evaluate how time was actually being

0740-7459/94/$04.00 © 1994 IEEE

spent. We found both techniques useful in quantifying the effect of social processes, and we gained several significant insights about existing processes.

SPECIAL MEASUREMENT CONCERNS

In both experiments, we studied software-development processes in action by observing what people do. We encountered several special issues:

- Protecting the anonymity of study participants, or subjects, and minimizing interference with their work.
- Documenting the the study site.
- Selecting the study sample.
- Choosing the instrumentation and levels of resolution.

Experimenting with people who work. People being observed as part of a study have understandable concerns about how the information will be used and who will see it. Anonymity and confidentiality are of the utmost importance — careers may even be at stake! You must also ensure that the study does not interfere with normal work. We were aware that some people might be uncomfortable about participating in our experiments, so we spent considerable time beforehand explaining the purpose of the studies. We reminded the subjects that there are no right and wrong ways to work; our purpose was not to judge but to understand behavior within a given environment.

We entered all data under an ID code known only to the researchers. We also gave each subject a list of rights, including the right to temporarily discontinue participation at any time, to withdraw from the study, to examine the research notes, and to ask us not to record something. Not one of our subjects exercised these rights, but it made them more comfortable knowing they were there.

Documenting the study site. Software-development organizations come in many shapes and sizes with distinct cultures and vastly different products. To make it easier to compare results across studies, researchers must get in the habit of clearly delineating the principle dimensions of the organization, its process, and the supporting technology.

The subjects in our experiments build software for a real-time switching system consisting of more than 10 million noncommented lines of C code, divided into 41 subsystems. New hardware and software functions are added every 15 months or so.

Project management was interested in tracking features, the units of functionality a customer will pay for. Feature size varies from a few lines of noncommented code with no new hardware required to 50,000 lines of noncommented code with many complex, specially designed hardware circuits. Generally, a developer works on no more than two features at once. Most software is built using a real-time operating system.

DATA ON REAL USERS, EVEN IF THE SAMPLE IS SMALL, IS REVEALING.

The organization responsible for product development has approximately 3,000 software and 500 hardware developers. It is registered to the International Organization for Standardization's ISO 9001 standard and has been assessed at level 2 on the Software Engineering Institute's Capability Maturity Model.

The purpose of documenting an organization's dimensions is to provide boundaries for the results of the study. We are not attempting to claim that our results generalize to all organizations. On the contrary, the relative size, complexity, and maturity of this software system are inextricably associated with its process.

Selecting the sample. A trade-off always exists between minimizing the possible variance and maximizing the ability to generalize a study's findings. On the one hand, you want results to apply to many settings and samples. On the other, you need to control the boundaries to eliminate the possibility of random errors and improve the validity of the results. The researcher must ultimately decide which extraneous variables are most important to control for, and there is no standard way to do this. You must rely on judgment, study prior research, and determine available resources. One way to avoid facing these control issues until late in the study is to collect as much information as possible on the sample subjects and then control for various effects in your analysis.

We applied a pupposive sampling scheme — in which subjects or cases typical of the target population are selected — choosing subjects at random yet stratifying along significant dimensions. We felt that project (organization, phase, and type) and personnel (age, gender, race, individual personality, and years of experience) factors were the most important to control for. Our goal was not to conduct a comparative study but to obtain a broad base of observations, thereby decreasing the likelihood of idiosyncrasies in our findings.

We realize that our sample sizes are small and probably inadequate for statistical validity but, as Fred Brooks pointed out "any data is better than none."[1] This work falls within the second category of nested results Brooks cites as necessary and desirable for progress in software development: reports of facts on real user behavior, even though that behavior is observed in undercontrolled, limited samples.

Choosing instrumentation and resolution level. Finally, there is the question of instrumentation. How do you get the data and at what resolution? Resolution — the level of analysis and frequency of sampling — is a fundamental concern in

any decision to measure processes and determine cost-benefit trade-offs.

As we mentioned earlier, our goal was to understand the structure of software-development processes. To accomplish this, we used a model that views development processes as complex queuing networks. In most software projects, there is a large discrepancy between *race time* — time spent in actual work — and *elapsed time*. Race time expands to elapsed time when there are interruptions, blocking (some obstacle to continuing work), and waiting periods. We wanted to document the factors that inhibited progress and their effect on overall development time. We considered several methods of data collection before settling on time diaries and direct observation.

A standard tool in behavioral science is the retrospective one-time survey questionnaire. However, this method tends to provide a relatively flat, static view of the development process, and we were interested in the dynamic behavior of people performing highly interdependent tasks. Hence we chose to design a modified time card — a time diary — and asked developers to record their daily activities over 12 months.

To calibrate the accuracy of the time diaries, we needed an instrument that would give us a finer resolution than was possible with time diaries. One option was video cameras. Although there are experimental precedents, we felt it would be inappropriate because our study subjects would not be used to such intrusiveness. While they were fairly receptive to the notion of participating in experiments, the introduction of video equipment would have distorted their behavior (not to mention that of their peers and overall work progress). Also, we would have to watch and interpret more than 300 hours of videotape, which would add both cost and time to the experiment. Finally, we were interested in getting data about *why* developers used their time the way

WHY MEASURE PEOPLE FACTORS?

In 1975, Fred Brooks described software construction as "an exercise in complex (human) interrelationships."[1] Yet almost 20 years later, most improvement exercises are still focusing on the technological aspects of building software systems — the tools, techniques and languages people use to write the software. Relatively little systematic study has been done about the associated people issues: how to ensure accurate and effective communication about a product no one can see, how to maintain project motivation and how to keep focused despite obstacles and distractions. Although many articles have addressed the importance of such issues, few have conducted systematic investigations about their operation in software development.

We believe there are three reasons to emphasize the organizational and social context of software processes.

- Even in the most tool-intensive parts of the process, such as the system-build process, the human element is critical and dominant. While the efficiency and appropriateness of the tools are obviously important, the crucial job of tracking down sources of inconsistency and negotiating their resolution is performed by people.
- Numerous process studies have indicated a large amount of unexplained variance in performance, suggesting that significant aspects of the process are independent of the technology. You could even argue that continually introducing more and more sophisticated tools is responsible for what Brooks calls the "tool mastery burden,"[1] which itself contributes to performance variance. Moreover, all too often tools are designed in isolation and subsequently fail to achieve their promised potential once in use. Genuine advances in tools and languages must be accompanied by a consideration as to how the technology will be incorporated into the existing social and organizational infrastructure.
- Several prominent authors have noted that much of a project's effort is devoted to issues outside programming; both Brooks and Barry Boehm estimate that as much as half a programmer's time is absorbed by machine downtime, meetings, paperwork, and miscellaneous company business.[1,2] If only half the time is spent programming, and technical advances are not making a big difference in productivity, perhaps we need to look elsewhere for ways to improve the development process.

These three reasons give rise to the need for new measures, iterative exercises that increase understanding through the use of significant measures, and the recognition that this iteration is a prerequisite to assessing and justifying process improvements.

More effective measures. Because software development yields a collaborative, intellectual — as opposed to physical — output, techniques to measure it must be both creative and carefully considered. In addition, the pace of technological and market changes and constantly shifting organizational structures, mean that we must regularly reevaluate our assumptions about development processes. A changing environment may render old assumptions invalid. For example, a narrow technical focus can generate many myths, such as "developers don't like to be (and, hence, cannot be) observed" and "programming is an isolated

they did — why they made certain choices and how they decided among competing demands on their time. With video, we would not have been able to ask the subjects about their choices as they were making them. Given these drawbacks, we decided to use direct observation — we watched a sample of participants as they worked.

Figure 1 illustrates how the resolution of the time diary contrasts to that of direct observation. Observer-recorded data contains an impressive amount of micro-level detail, often down to three-minute intervals. To effectively compare it with time diaries, we summarized that detail into major activity

Diary		Observer's notes	
0800-1800	Working on high-level design	0800-0900	Administration
		0900-1010	High-level design analysis
		1010-1021	Break
		1021-1135	Code experiment with peer
		1135-1226	High-level design document writing
		1226-1314	Lunch in cafeteria
		1314-1330	Answer document question (responsible person out)
		1330-1349	Answer growth question
		1349-1406	Reading results of Business Unit Survey
		1406-1500	Code experiment with peer
		1500-1626	Searching for paper and reading
		1626-1701	Code experiment with peer
		1701-1705	Administration

Figure 1. *Sample comparison sheet comparing a software developer's self-reported time diary with the observer's notes. This sheet is typical of the calibration. The difference in end time between the diary and the observer's notes is approximately 55 minutes. The diary contains one entry for this nine- to 10-hour day. The observer, on the other hand has 13 entries, about five hours of which correspond to high-level design activities.*

activity."[3]

Another reason for regular reevaluation is that there are not enough studies that address practical ways to handle new problems. Most studies that investigate programming's human aspects rely primarily on student programmers or artificial tasks in laboratory settings.[4] Although these studies are informative, we question how useful they are in large-scale development. How representative are the samples and tasks? What kinds of problems unique to organizational environments are being ignored by this focus on small and artificial domains?

Iterative exercises. Answering the question of what to measure is an iterative exercise that increases your understanding of the process and helps you transform that understanding into practical steps toward improvement. As a performer and observer of processes, you have some intuition about where problems lie. For example, if you see that progress is often blocked, the obvious thing to measure is what is blocking it. If meetings appear to be significantly impeding progress, it is logical to measure the number, duration, and effectiveness of meetings to understand their effect on process and performance.

This is the beginning of the iterative exercise. You use your understanding to determine what measures to take and then use the results of those measures to confirm or deny your hypotheses. An important part of this exercise is not letting preconceptions interfere with the possible insights to be gained from the measurements.

How basis for improvements. Many process-improvement claims are based on anecdotal evidence or reasonably plausible arguments. While these may give some comfort, they do not constitute a quantifiable basis for claiming improvement. The understanding of processes musts be firmly rooted in measurements. This solid basis lets you accurately benchmark existing processes and quantify the value of subsequent improvement efforts.

A significant precedent for such an approach is the work done in the early 1960s in Japanese software factories, where Japanese developers gathered data on existing processes before changing or improving them.[5] More recently, Alexander Wolf and David Rosenblum noted that to improve processes and design new ones, you must first obtain concise, accurate, and meaningful information about existing processes.[6] That is, by understanding how and why programmers work the way they do, we will be better positioned to identify tools and methods that enable them to perform tasks better and in less time.

REFERENCES

1. F. Brooks, Jr., *The Mythical Man-Month*. Addison-Wesley, Reading, Mass., 1975.
2. B. Boehm and P. Papaccio, "Understanding and Controlling Software Costs," *IEEE Trans. Software Eng.*, Oct. 1988, pp. 1462-1477.
3. G. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
4. W. Curtis, "By the Way, Did Anyone Study Any Real Programmers?" in *Empirical Studies of Programmers*, R. Soloway and S. Iyengar, eds., Ablex Publishing, Norwood, N.J., 1986, pp. 256-262.
5. M. Cusumano, *Japan's Software Factories*, Oxford University Press, New York, 1991.
6. A. Wolf and D. Rosenblum, "A Study in Software Process Data Capture and Analysis," *Proc 2nd Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 115-124.

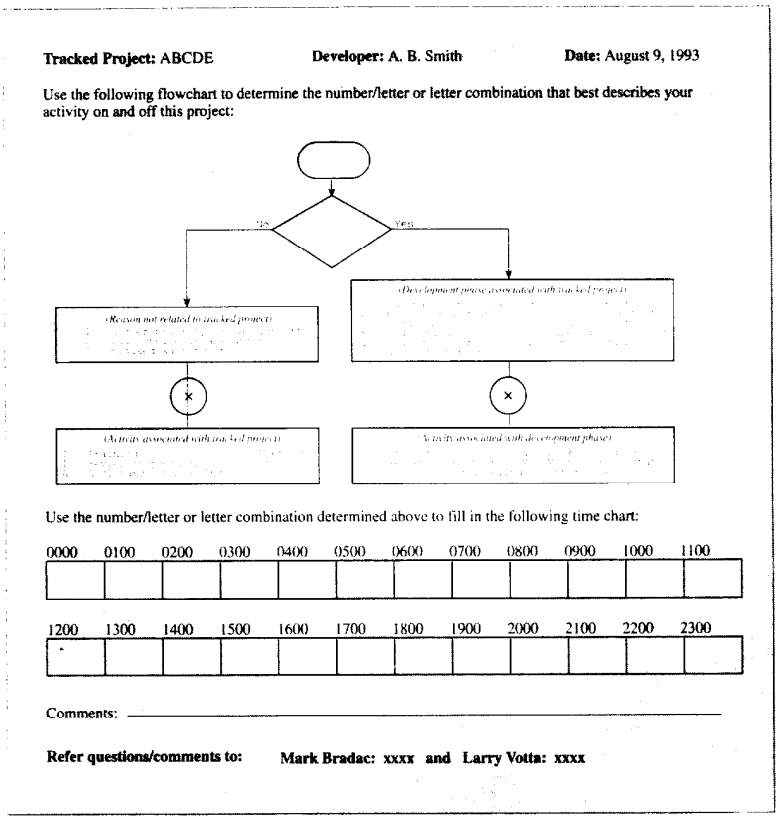


Figure 2. *Self-reporting time form for software developers.*

blocks. We then verified the reliability of the summary process by randomly comparing reports prepared by independent researchers. The level of comparability was well within accepted research standards.

By following this approach, we were able to validate the time diary as a low-cost, effective way to determine how people spend their time. This, in turn, served as a mechanism to obtain coarse-grained data about the software process, because processes are ultimately what make people do things.

TIME-DIARY EXPERIMENT

Before we conducted this experiment, we did an initial pilot study, drawing on one programmer's personal log to construct an initial time-diary instrument. The log let us identify the principal activities and working states as well as formulate several hypotheses, which we tested in subsequent experiments. For this discussion, we focus on the experiment itself. The complete details of the pilot study are presented elsewhere.[2]

Over the one-year life of the experiment, 13 people from four software-development departments filled out the time diary on a daily basis. During the study, we revised the time-diary form several times as a result of both positive and negative feedback from the subjects.

Figure 2 illustrates the final time-diary form. It is easier to use than our initial version (most subjects spent two to three minutes daily filling it in, as opposed to five to 10 minutes a day for the earlier form), yet it still managed to capture the basic data we needed about development processes. The resolution of reported time was one-hour segments — a relatively large granularity, but one appropriate to our goals. All we had to know for each time segment was if the developer was working on the assigned feature. If he was, then we simply had to know the appropriate task within the process and the appropriate activity within that task. (We extracted task steps from a definition of the development processes; activities partition the possible ways in which a developer may be performing that task.)

If the developer was not working on the assignment, we had to differentiate among reasons for not working: reassigned to a higher priority project; blocked, waiting on resources; or personal choice to work on another activity.

Figure 3 presents the distribution of time spent over various development tasks (subjects generally rounded off time to the nearest half hour). Even though the development phase was primarily coding, there is a reasonable distribution of time spent on other tasks. In fact, roughly half the time is occupied by noncoding tasks. This indicates rather clearly that not only does the waterfall model not reflect what actually goes on (which every developer already knows), but the accepted wisdom of an iterative and cyclical model of development is also inadequate. In a large project such as this, both product and process are in multiple states at once. We have, in essence, many iterative, evolutionary development processes being performed concurrently.

Figure 4 shows the distribution of time over process states (working on the process, blocked and waiting for resources, not working on the process). The ratio of elapsed to race time is roughly 2.5 — developers worked on a particular development only 40 percent of the time. They spent the rest of the time either waiting on resources or doing other work.

We learned from our pilot study that blocking varies throughout the development cycle, and that coding often exhibits the least amount of blocking. We surmise that this reflects the low dependency on outside organizations, resources, and experts during this phase. (We later corroborated this result in the direct-observation experiment.) We also discovered that most of the developers were working on two development projects at once, which we believe is the way organiza-

tions choose to deal with blocking.

The amount of rework done was about 20 percent — roughly one fifth of the total working time. The time spent not working is clearly dominated by reassignment to other projects. This reassignment emphasizes two important aspects of large-scale software development. First, project organization is extremely dynamic because priorities change and requirements evolve. Second, this is a real-time system that is being simultaneously used and modified, so in addition to fielding occasional critical customer problems, developers must customize new features for specific customers.

Figure 5 presents a histogram of the duration of time intervals across all study subjects. The intervals tend to be clustered in common patterns. The significant number of four-hour working segments reflects the day broken in half by lunch. The frequency of two-hour segments is due to the organizational mandate limiting review meetings to two hours or less. The significant number of eight-hour segments is due to the test laboratory being scheduled for either four- or eight-hour intervals, depending on the complexity of the lab setup.

DIRECT-OBSERVATION EXPERIMENT

Although periodic interviews and occasional unannounced visits had convinced us that no conscious misrepresentation was occurring, we wanted to check the time diaries in a second experiment using direct observation. Although direct observation and ethnographic studies are fairly common in social science research, they are highly unusual in studies of software development. A common rationale is that "software developers don't like to be (and therefore cannot be) observed." The truth is that *no one* likes to be observed. However, a well-designed experiment can do much to alleviate trepidation,[3] and such an approach is

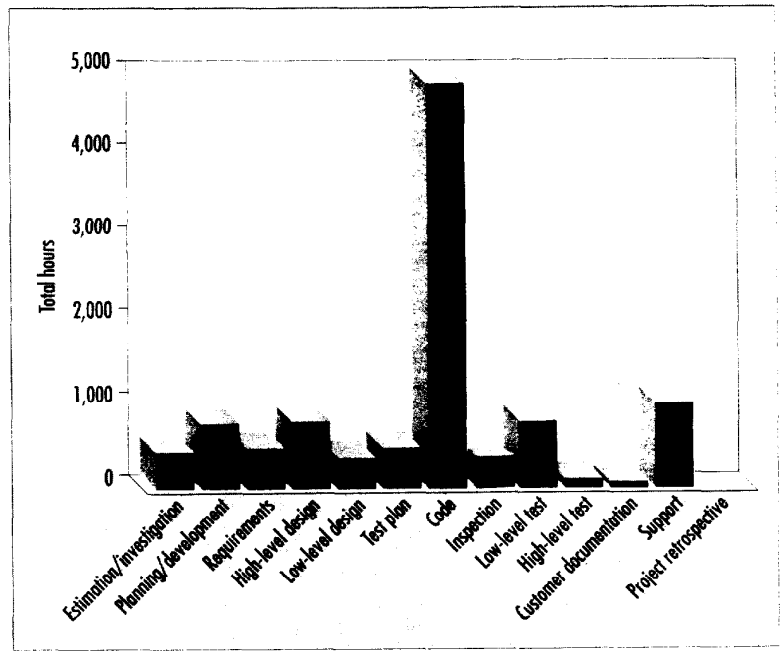


Figure 3. *Self-reported total time by task. This histogram shows the number of self-reported time-diary hours for each process task for all subjects. The development organization is functionally organized, and participants for this cross-sectional study were drawn from four software-development groups. Only about half the hours are reported as coding.*

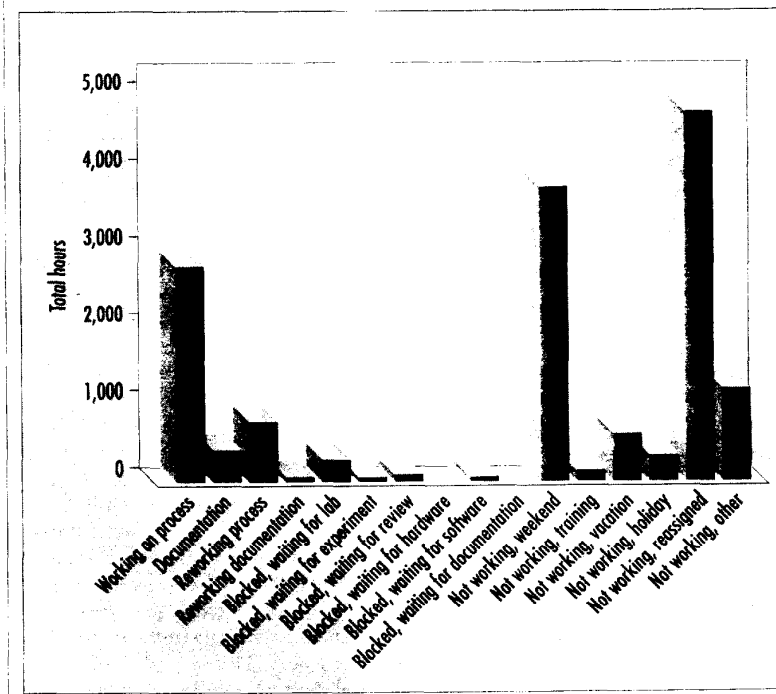


Figure 4. *Self-reported time diary. This histogram shows the number of self-reported time-diary hours for each process state for all subjects. The maximum amount of time is reported in the Not working, reassigned state. This reflects the dynamic nature of prioritizing work on developing features in this organization.*

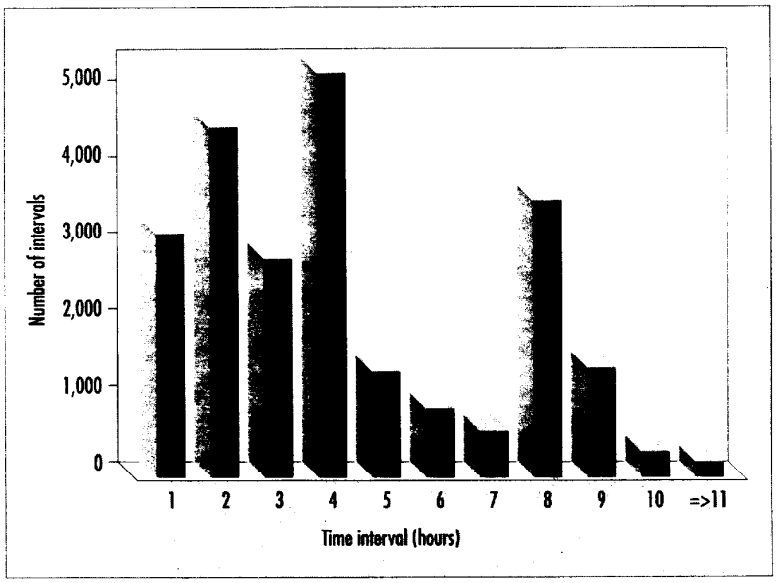


Figure 5. Self-reported time-diary intervals. This histogram shows the number of self-reported time-diary entries by all subjects. There were over 600 diary entries of from four to five hours. The histogram demonstrates that, even in a relatively crude measurement like a self-reported time diary, there are recognized breaks in the use of time. The spikes at two, four, and eight hours are part of the culture of the development organization. The two-hour peak occurs because the maximum review meeting time is two hours. The four-hour peak is the break caused by lunch. The eight-hour break occurs when the subject either worked through lunch or only had one entry per day.

not entirely without precedent, as the box on pp. **AA-AA** describes.

We randomly chose five software developers from the group participating in the time-diary experiment. We also included two software developers outside the self-reporting experiment so that we could evaluate the effect of self-reporting plus observation and observation without self-reporting. Somewhat surprisingly, no one who was asked refused to participate.

Procedures. We observed each subject for an average of nine to 10 hours per day for five days over 12 weeks. Each subject chose and scheduled two of the five days. Interestingly, subjects often forgot when they had scheduled such sessions and were surprised to see the researcher in the morning — further proof that they were not too intimidated by the prospect of being observed. The remaining three days were assigned by random draw without replacement, and the subjects were not informed of when they would occur.

The logistics behind this were not trivial. For example, vacations had to be blocked out in advance, and the observer had to adjust her schedule to accommodate subjects who worked flexible hours. Many lab sessions were also conducted off-hours, and we had to establish what to do if a developer did not come into work.

We used continuous real-time recording for nonverbal behavior and interpersonal interactions. When a developer was working at the terminal, we used a time-sampled approach, asking the developer at regular intervals "What are you doing now?" We recorded daily observations in small spiral notebooks, one for each subject. Each evening, we converted the notebook observations to standard computer files. This let us readily fill in observations while they were still fresh and served as the basis for interim summary and analysis sheets. As data came in, we added it to a loose-leaf notebook, with separate sections for each subject. This helped us stay organized over time and made it easier for the researchers to communicate.

Insights. The direct-observation study gave us several significant insights about the process, including the effect of communication on work flow and the use of communication media.

Communication and work flow. Gerald Weinberg once posed the provocative question, "Does it matter how many people a software developer runs into during the day?"[4] He argued that although the task of writing code is usually assigned to an individual, the end product will inevitably reflect the input of others. Indeed, we were impressed by the amount of time each developer spent in informal communication — on average, 75 minutes per day of unplanned interpersonal interaction (although this was scattered into episodes of widely differing duration).

Organizational theorists have long acknowledged that information flow is critical to an organization's success.[5] Most studies of communication in collaborative work, however, have a narrow focus, typically restricting results to one communication medium or focusing on exchanges that were planned and relatively long. Moreover, the empirical data often consists of asking subjects who they talk to the most, which risks confusing frequency with duration or effect. Our study, on the other hand, tracked all communication activity, at the individual level and across four media: e-mail, phone, voice mail, and in-person visits. We did not include paper because hard-copy documentation is practically nonexistent in this organization; all documentation is kept up to date on line.

For each subject, we kept a summary sheet of what we observed their interactions to be across the four media. The interactions were on-the-fly exchanges usually involving little formal preparation and little reliance on written documentation, diagrams, or notes. We broke down each interaction in terms of whether it was sent or received by the subject.

We discovered two interesting things:

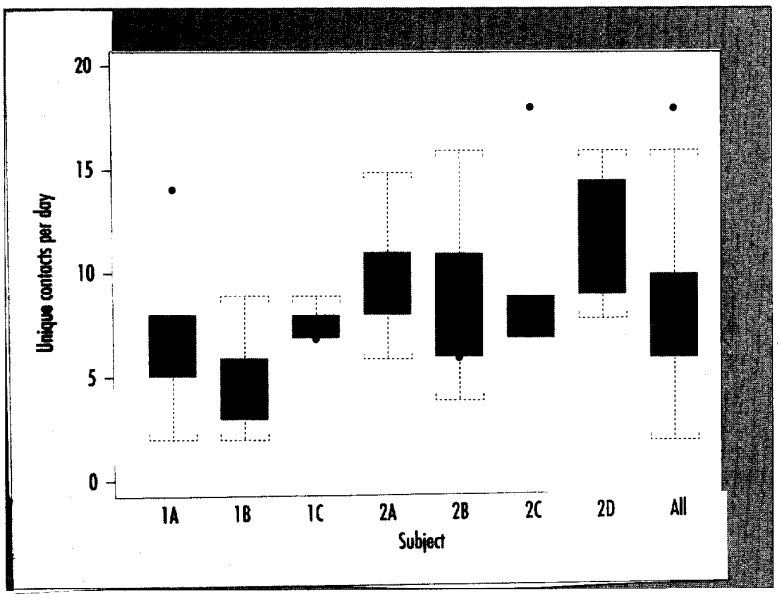


Figure 6. Unique contacts per subject per day. This figure reflects interactions across four communication media: voice mail, e-mail, phone, and personal visits, but does not include contacts made during meetings or lab testing. It also does not include purely social exchanges. The median over all subjects is 7 (last boxplot). The outliers reflect primarily days in which a subject was modifying existing code.

♦ There was much unplanned interaction with colleagues: requests to informally review code, questions about a particular tool, or general problem-solving and debriefing sessions.

♦ Former colleagues made up a surprising percentage of the contacts. One of our subjects who had transferred to another department approximately two months earlier received, on average, one call a day from his former group. (Of course, we would expect this trend to decline over time for a particular individual.)

We did not include contacts made in (scheduled) meetings or in the laboratory, purely social exchanges, or exchanges with "faceless" administrators.

Number of unique contacts. Figure 6 is a box-plot diagram showing the number of unique daily contacts over five days of observation for each subject. A box plot is an excellent and efficient way to convey certain prominent features of a distribution. Each data set is represented by a box, the height of which corresponds to the spread of the central 50 percent of the data, with the upper and lower ends of the box being the upper and lower quartiles. The dot within the box denotes the data median. The lengths of the vertical dashed lines relative to the box indicate how stretched the tails of the distribution are; they extend to the standard range of the data, defined as 1.5 times the interquartile range. The detached points are outliers. As depicted by the far right box plot, the median number of unique contacts, across all study subjects, was seven per day.

The outliers are particularly interesting. The highest point (17 unique contacts) represents a day in which developer 2C started to work on a code modification motivated by a customer field request. The other outliers also correspond to modifications of existing code, and in each case, the number of unique interfaces approximately doubled from the baseline of seven. Most of these contacts were requests for authorization to change code owned by another developer. Just slightly less frequent were calls to a help desk for passwords or information about a particular release, calls to the lab requesting available time slots for testing, and exchanges with peers about process procedures in general.

These contacts were not technically related per se. That is, the solution was often not the motivating issue driving this behavior. Rather, the developers needed help implementing the solution.

Frequency of communication. Figure 7 shows the number of messages being sent and received each day across the different media. The distributions of sent and received visits and phone messages are both approximately normal, reassuring us that the sample is not significantly skewed and also suggesting the presence of reciprocal interactions. (We did not explicitly track communication threads, a group of related communication events devoted to a single problem.[6])

As the far right set of boxes shows, a developer typically received a total of 16 messages and sent a total of six messages each working day. Ignoring e-mail for the moment, the most ubiquitous form of contact in this work environment was in-person visits. They occurred approximately two to three times as often as the other media.

Asymmetry of e-mail use. One of the most surprising results was the use of e-mail. Many corporations are starting to implement this new form of communication, and we fully expected, given the computer-intensive nature of this organization, to see a large amount of e-mail traffic. However, although our subjects received many such messages (a median of nine per day), they sent very few (a median of zero per day). What's more, the content of these messages was rarely technical. Most of the traffic was devoted to organizational news (announcements of upcoming talks, recent product sales, and work-related congratulatory messages) or process-related information (mostly announcements of process changes).

We attribute this phenomenon to several factors.

♦ It is difficult and time-consuming to coherently draft a complex technical question or response. As noted by one developer "E-mail is too slow; by the

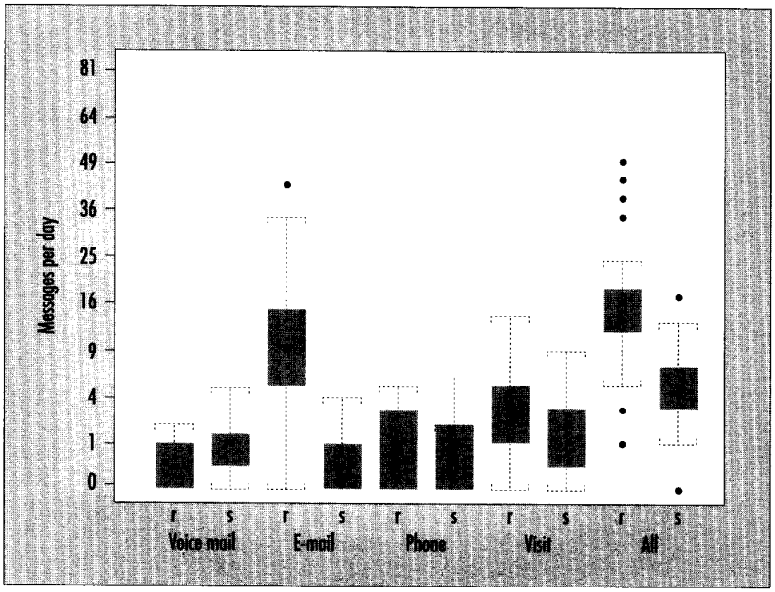


Figure 7. *Messages sent and received across four media types. The figure shows the number of messages sent and received, by media type and according to whether they were received (r) or initiated (s). We applied a square-root transformation to stabilize the variance. Each box contains data on all seven study subjects across five days of observation per subject.*

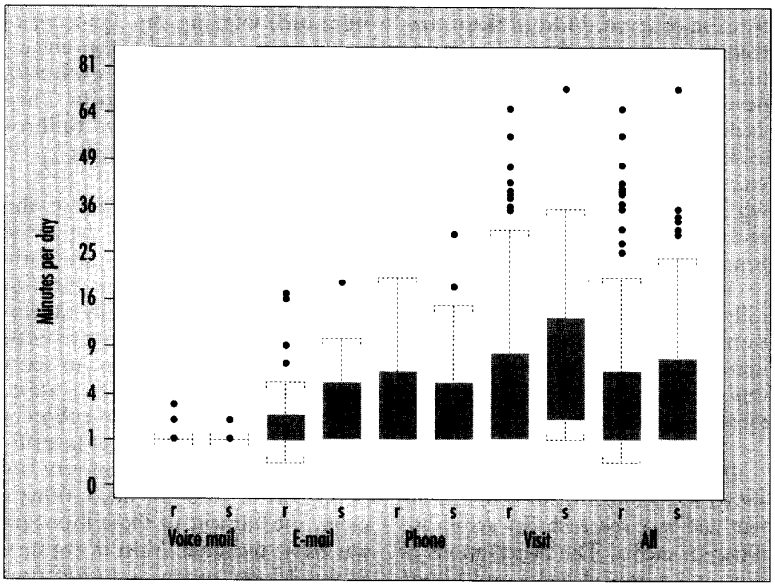


Figure 8. *Duration per contact by media type. The duration per contact is broken down by media channel and according to whether the message was received (r) or initiated (s). We applied a square-root transformation to stabilize the variance. Each box contains data on all seven study subjects across five days of observation per*

time I type out a coherent description of the problem, I could have called or walked over and gotten the answer."

♦ The ambiguity of software technology may necessitate a type of iterative problem-solving that is ill-suited to the e-mail venue. Then again, our subjects may have been reluctant to release a written recommendation or opinion without having control over its final distribution.

♦ The e-mail system being used had been around for 10 years, long enough for a use pattern to emerge. In this organization, e-mail appears to be synonymous with broadcasting. The flooding of the system with nontechnical messages may make developers reluctant to use it for pressing technical issues.

Length of communication. Figure 8 plots the duration of messages in each medium. Looking across all forms of communication, approximately 68 percent of the interactions are less than five minutes long. This agrees with research done in the early 1980s at Xerox Palo Alto Research Center[7] on a somewhat larger sample. It also confirms anecdotal evidence supplied by independent studies of this population.

Predictably, voice-mail messages are very brief — one minute. Surprisingly, phone conversations are also brief — two to three minutes. Both require the same amount of time to receive as to send.

The figure also shows that subjects needed less time to read an e-mail item than to send one. This is also not surprising, since composing a satisfactory message requires more thought than reading one. In the same vein, it takes about three minutes longer to make a visit than to receive one because the visitor must walk to the desk of the person being visited.

Finally, there are significant outliers in all media; visits of close to one hour and phone calls of 30 minutes are not uncommon. This result is particularly nonintuitive because all these interactions are unplanned and unanticipated.

Our primary motivation for this study was to measure and understand aspects of process intervals, but we also wanted to investigate underdeveloped arenas in software research: the social structure, environment and culture of a real organization of software developers. It is our belief that all three elements

(organization, process and technology) must be addressed before we can get a complete picture of the development process.

The results support our belief that elements of the organization are as important as technology, if not more so: A large percentage of the process cycle was devoted to aspects not directly involving technology, and the data on the number of interpersonal contacts a developer requires during a typical working day strongly suggests that technical problems are not the real issue in this organization. Rather, these software developers need to apply just as much effort and attention to determine who to contact within their organization to get their work done.

Most important, we were able to quantify what had previously been predominately qualitative impressions about life as a software developer in this firm:

- People are willing to be observed and measured if you take the proper precautions.
- Software development is not an isolated activity. Over half our subjects' time was spent in interactive activities other than coding, and a significant part of their day was spent interacting in various ways with coworkers.
- Progress on a particular development is often impeded for a variety of reasons: reassignment to a higher priority task, waiting for resources, and context switching to maximize individual throughput.
- On average, work is performed in two-hour chunks.
- Time diaries are adequate for their intended level of resolution. What is missing, however, is data on unplanned, transitory events. Direct observation showed us that developers spend about 75 minutes per day in unplanned interpersonal interactions.
- There are seven unique personal contacts per day on average, representing continuing interactions; this can double for certain kinds of activities.
- Direct interpersonal communications are the dominant means of interaction. E-mail tended to be used as a broadcast medium in this organization rather than as a means of exchanging technical ideas or achieving social consensus. This fact is particularly important as much of cooperative-work technology presupposes e-mail as the central basis for cooperation.

Motivated by the three surprising results — the expansion factor of race time to development time, the amount of unplanned interruptions, and the limited use of e-mail — we are now building queuing models to help us understand development time. At the same time, we are continuing to investigate the functions that meetings, both planned and unplanned, serve in the process. ◆

ACKNOWLEDGEMENTS

We are grateful for the extremely hard work of our collaborators, who made all these experiments possible: Mark Bradac, Dale Knudson, and Dennis Loge. Tom Allen of MIT brought us together, and Peter Weinberger, Eric Sumner, and Gerry Ramage provided the financial support. Marcie Tyre of MIT was particularly helpful, providing extensive input along the way and pointing us in the direction of relevant organizational theory. Bill Infosino's early suggestions regarding experimental design issues and Art Caso's editing of an earlier version of this article are also much appreciated.

Finally, we acknowledge the cooperation, work, and honesty of the study subjects and their colleagues and management for supporting our work with their participation.

REFERENCES

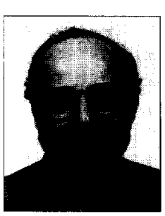
1. F. Brooks, Jr., "Plenary Address: Grasping Reality Through Illusion," *Proc. Computer-Human Interface Conf.*, ACM Press, New York, 1988, pp. 1-13.
2. M. Bradac, D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, 1993, pp. 155-165.
3. C. Judd, E. Smith, and L. Kidder, *Research Methods in Social Relations*, 6th ed., Harcourt Brace Jovanovich, New York, 1991.
4. G. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
5. T. Allen. *Managing the Flow of Technology*, MIT Press, Cambridge, Mass., 1977.
6. A. Wolf and D. Rosenblum, "A Study in Software Process Data Capture and Analysis," *Proc. 2nd Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 115-124.
7. M. Abel, "Experiences in an Exploratory Distributed Organization," in *Intellectual Teamwork*, J. Galegher, R. Kraut, and C. Egido, eds., Erlbaum Publishing, Hillside, N.J., 1990, pp. 489-510.

Dewayne E. Perry is a member of the technical staff in the Software and Systems Research Center at AT&T Bell Laboratories. His interests include software-process descriptions, analysis, modeling, visualization, and environmental support, as well as software architectures, software-development environments, and the practical use of formal specifications and methods. He is on the editorial board of *IEEE Transactions on Software Engineering*.

Perry received an MS and a PhD in computer science from Steven's Institute of Technology. He is a member of the ACM and IEEE.

Nancy Staudenmayer is a doctoral student in management of technological innovation at MIT's Sloan School of Management. Her interests include the management of software-development projects and how new forms of information technology affect organizations.

Staudenmayer received a BA in mathematics from Wellesley College and an MA in statistics from the University of California at Berkeley.



Lawrence G. Votta is a member of the technical staff in the AT&T Bell Laboratories' Software Production Research Department at Naperville, Illinois. His interests include understanding how to measure and model large and complex software-development processes.

Votta received a BS in physics with high honors from the University of Maryland at College Park, and a PhD in physics from the Massachusetts Institute of Technology. He is a member of the IEEE Computer Society and ACM

Address questions about this article to Perry at AT&T Bell Laboratories, Rm. 2B-431, 600 Mountain Ave., Murray Hill, NJ 07974; dep@research.att.com.