An Examination of Software Engineering Work Practices[1]

Janice Singer[α], Timothy Lethbridge[β],
Norman Vinson[α], Nicolas Anquetil[β]

[α] Institute for Information Technology
National Research Council, Ottawa, ON, K1A OR6

[β] School of Information Technology and Engineering
University of Ottawa, Ottawa, ON, K1N 6N5

Abstract

This paper presents work practice data of the daily activities of software engineers. Four separate studies are presented; one looking longitudinally at an individual SE; two looking at a software engineering group; and one looking at company-wide tool usage statistics. We also discuss the advantages in considering work practices in designing tools for software engineers, and include some requirements for a tool we have developed as a result of our studies.

1. Introduction

The Knowledge Based Reverse Engineering project's goal is to provide software engineers (SEs) in an industrial telecommunications group with a toolset to help them maintain their system more effectively. To achieve this goal, we have adopted a user-centered design approach to tool development [6, 7, 8]. However, unlike traditional user-centered approaches, we have focused on the SEs' *work-practices*. This represents a new approach [15] to tool design.

This approach borrows from several different fields in an effort to more accurately assess users' behavior and then provide them with tools that enhance, rather than displace or replace, these work practices. The rationale is that the tools that are built will actually be used because they have been created to mesh with existing behavior. This paper will describe our experiences with this approach and what we have learned about the work practices of one group of SEs at a large telecommunications company.

The rest of this introduction will first critically examine the more traditional uses of psychology in the program comprehension literature, and second describe the study of work practices. We will then outline some results of a study we conducted at a large telecommunications company. Finally, we will discuss the implications of these results for tool design .

1.1 Empirical Studies of Programmers (ESP)

One human-computer interaction approach to the design of tools has been to study the cognitive processes of programmers as they attempt to understand programs [19, 20, 21]. The results of such studies are supposed to provide the basis for designing better tools. In other words, understanding the mental processes involved in programming will permit the design of tools that mesh with the programming process.

In this vein, ESP research has identified a number of programmers' approaches to the comprehension 'problem' including the top-down [12], bottom-up [4], and as-needed strategies [9], and the integrated meta-model [21].

There are three problems with this research, though, as it pertains to tool design. First, the vast majority of the research has been conducted with graduate and advanced undergraduates serving as expert programmers (but c.f., [21]). It is not at all clear that these subjects accurately represent the population of industrial programmers. Conse-

[1] This work is supported by NSERC and the company serving as the site of the study.. This work was sponsored by the Consortium for Software Engineering Research (CSER). The IBM contact for CSER is Patrick Finnigan.

quently, the results of studies involving students cannot be generalized to programmers in industry.

Second, to control extraneous variables, researchers have used programs that are very small (both in terms of lines of code and logic) relative to industrial software. This poses a generalization problem as well: it is not clear that approaches to comprehending small programs scale up to the comprehension of very large programs.

Third, there is an assumption that understanding the programmer's mental model is an efficient route to designing effective tools. However, it is not at all obvious how to design a tool given a specification of the programmer's mental model. For instance, how does knowing that programmers will sometimes use a top-down strategy to understand code [12] inform tool design? It doesn't tell us what kind of tool to build, or how to integrate that tool into the workplace or the programmer's work. Furthermore, given this knowledge, it is not clear how to help the programmer build that mental model; how to help her apply it; or how to help her use it effectively in software engineering activities.

These three problems with the ESP approach suggest that an alternative approach to tool design may be more effective.

1.2 Human Computer Interaction

Currently, there is a strong focus on usability in the field of human-computer interaction [10]. That is, designers attempt to ensure that prospective users can use the software without encountering interface difficulties. For instance, it should be clear to users what action they should take at each step, preferably without referring to documentation. Another aspect of usability is the minimization of the number of steps and the amount of time needed to accomplish a task. To determine whether software is sufficiently usable, prospective users are observed using the software for a few, or several minutes. Reaction times, errors, backtracking to previous states and failures to accomplish the task are recorded along with the conditions under which they occurred. These data are then used to fix the interface and, ideally, more of these test-redesign iterations take place until the software is sufficiently usable.

However, we see problems with this approach. While it may increase the *usability* of systems, it does not guarantee that the systems that are built will be genuinely useful [2, 18]. The usability approach cannot speak to the issue of whether a user will adopt and use a new tool in the workplace because that is not the point, or the focus, of usability. Moreover, several features of the usability approach prevent it from informing the designers about the acceptance of the tool in the workplace. Usability testing usually takes place outside the normal work setting, sometimes in a room especially designed for that purpose. This method of testing prevents the user from behaving in a normal manner because it isolates him from resources that are not part of the software (such as colleagues, documentation, notes). In other words, it prevents the user from engaging in his day-to-day work practices. In addition, during usability testing, the user is essentially forced to use the software. In consequence, it is impossible to collect data on whether the user *would* use the software *if* he were given a choice between his existing work practices and the new software.

The lack of tool adoption and use is a major problem in the area of tool design for software engineering. However, because of its features and techniques, usability cannot inform designers on this issue. We believe that to build tools that are actually used, designers must first understand what it is that SEs do when they work. This is the reason for our focus on work practices in designing software engineering tools.

1.3 Work Practices

The study of work practices is a relatively new field [2, 3, 18] which seeks to understand how work occurs and, from this understanding, suggest appropriate technologies for the workplace. Work practices have been studied in such diverse fields as law, navigation, document use, etc.

In studies of work practices, data are generally collected by following and recording the work that people do. Researchers often rely on ethnographic methodologies producing diverse sets of data. The challenge, then, is to take work practice data sets, and put them into a form that is useful to designers.

Our approach to this problem has been to implement many different data collection techniques[2] and see if the evidence from each converges. Then we will use these data to decide what types of tools would best solve the problems that SEs face in their daily activities.

The first thing that struck us when we entered the work place was that we did not know exactly what it was that the SEs did on a day-to-day basis. That is, we knew neither the kinds of activities they performed, nor the frequency with which these various activities took place. As far as we could tell, there were many hypotheses about the kinds of things SEs do, but no clear 'cataloging' as such of exactly how SEs go about solving problems. Consequently, we decided to begin our study of work practices by finding out what it is that SEs do when they do their work. First, we will briefly describe the characteristics of the workplace. Then, the rest of this paper will present the findings from several studies we conducted to answer this first question.

2. Workplace Characteristics

The group we are studying maintains a large telecommunications system that is one of the key products of the company. The management of the group is fairly informal, with group members able to select the problems on which they work.

Group members work in close proximity and often walk over to each other's desks with questions. The group also makes use of a laboratory in which the target hardware is installed.

2.1 The System

The system includes a real-time operating system and interacts with a large number of different hardware devices. The system contains several million lines of code with over 16000 routines in over 8000 files. It is also divided into numerous layers and subsystems written in a proprietary high-level language.

The system was first fielded in the early 1980s and has since been continually updated. Its importance to the company and its evolution are expected to continue for many years to come.

[2] These methods are detailed more precisely in [7, 8, 13]

Approximately 13 people actively work on various aspects of the system at the current time. Over 100 people have made changes to the source code during the life of the system.

2.2 Software Engineering Process And Tools In The Group

The group follows a well-defined process for creating new system features. They also keep detailed records of problem reports and the consequent changes to the system. Other important documents include the 'practices' that are followed by those who install and run the system in the field.

Careful attention is paid to quality control in the form of design reviews, informal code inspections, and an independent test team.

Development work is done on the Sun platform, although the SEs must also spend considerable time installing and running the software on various configurations of the target hardware.

3. SE Activities

We collected five basic types of SE work practice data. First, using a web questionnaire, we simply asked the SEs what they do. Second, we followed an individual SE for 14 weeks as he went about his work. Third, we individually shadowed 9 different SEs for one hour as they worked. Fourth, we performed a series of interviews with software

Activity	*% of people*
Read documentation	66%
Look at source	50%
Write documentation	50%
Write code	50%
Attend meetings	50%
Research/identify alternatives	33%
Ask others questions	33%
Configure hardware	33%
Answer questions	33%
Fix bug	33%
Design	17%
Testing	17%
Review other's work	17%
Learn	17%
Replicate problem	17%
Library maintenance	17%

Table 1: Questionnaire results of work practices (6 responses).

engineers. Finally, we obtained company-wide tool usage statistics. The next several sections will outline more precisely our methodologies and results from these various studies.

3.1 Questionnaire Study

We began this research by administering a web-based questionnaire. The questionnaire covered many different aspects of the SEs' work. Here we report their answers to a question on what they spend their time doing. Six SEs in the group of 13 responded. The question was open-ended, i.e., the SEs had to decide how to describe their work, rather than choosing certain activities from a list.

On average, SEs said that they spend 57% of their time fixing bugs, and 35% of their time making enhancements to the system. Table 1 shows more specifically the things they reported that they engaged in, and the percentage of people reporting that activity.

The most reported activity was reading documentation. SEs also reported that they spend time looking at source, writing documentation, attending meetings, and writing code. Other activities include consulting, both answering and asking questions, working with the hardware, testing, designing, and fixing bugs.

Because of the questionable validity of self-reports, we felt it was extremely important to not just rely on what SEs said they did, but to actually observe them as they worked. Hence the next sections of the paper describe two studies that we undertook towards this goal.

3.2 Individual study

We have been following one SE longitudinally from the time he joined the company (November, 1996). For the first six months, we spent about 1-1/2 hours per week with B. However, as B has become more expert, we have found that it makes more sense to meet once every 3 weeks. This is both because new things happen less frequently (e.g., experience with a new tool) and because B is more busy with 'real' tasks. B is an experienced SE (was previously a team-leader), thus while he is new to the company, he is certainly new to neither maintenance nor telecommunications software.

Our sessions with B consist of 3 distinct components. First we talk about what has transpired since the last time we met. This could be anything from code review to learning about a new tool to reading documentation, etc. Second, we ask B to look at a diagram of the system he previ-

Activity	***Description***
Call trace	Looking at an execution trace of the program
Consult	Either being consulted or consulting someone else
Compile	Linking or compiling a program
Configuration Mgt	Entering and using the in-house configuration management system (sometimes for updating, and sometimes to search for past updates)
Debug	Using either the high-level or low-level debugger
Documentation	Looking at documentation
Edit	Actually making a change to source code
Management	General software activities, such as meetings, code reviews, etc.
In-house tools	Using one of the in-house tools, primarily static software analysis tools
Notes	Taking notes, or reading past notes
Search	Using Grep, in-house search tools, or searching in an editor
Source	Looking at source code using editors or code viewers
Hardware	Interacting with the hardware, e.g., loading software, running software, configuring the hardware, etc.
Unix	Issuing a general Unix command such as ls, cd, etc.

Table 2: Categories of activities observed when shadowing software engineers

ously constructed and ask him to modify it if it does not reflect his current understanding. Finally, we 'shadow' B as he works for 1/2 hour. In this paper, we report the data from the shadowing.

3.2.1 Method

3.2.1.1 Subject

B has worked in the software industry for many years. Prior to joining the telecommunications company, he worked as a team leader for a nearby competitor. There, B maintained a product in the same category as the current product, but developed on a much smaller scale.

B has experience in several languages, but prior to joining the company, considered himself to be an expert only in an in-house proprietary language. Likewise, while he has experience in several platforms, prior to joining the company, B considered himself to be an expert only in an in-house proprietary 68K development platform. B has worked on 5 different systems, 3 of which have involved development, 2 of which have involved maintenance.

B joined the company in November, 1996. Before then B had no experience in the company's in-house Pascal-based proprietary language. Nor did B have any experience in Pascal, although he had programmed in other structured languages. B had utilized VI before coming to the company, but planned on switching to the Emacs editor at the company. Similarly, he had used Grep previously, but was switching to use of Egrep and Fgrep at the company. B did not have previous experience with the other tools available at the company

3.2.1.2 Procedure and Data

The shadowing data result from 14 half-hour sessions ranging from October 17, 1996 to February 27, 1997. Some days are missing because of vacation or schedule conflicts. For the most part, however, these dates reflect weekly meetings with B.

For half an hour, we would sit behind B and write down the things he did. For instance, if he used Grep, that would be recorded (using pencil and paper). If he read documentation, or wrote notes to himself, that was written down.

We recorded B's activities in detail, but not to the point of exactly what he typed or said. For example, we would record that B edited a file or interacted with the hardware, while not detailing the exact nature of his involvement with these activities.

A new activity was recorded each time a switch in activity occurred. So, for instance, if B did 6 Greps in a row, that was recorded as a single instance of the event Grep. Then if he did 4 Diffs, a single Diff event would be recorded. Taking that to its extreme, if all B did was Grep for 1/2 hour, that is the single activity that would have been recorded for that 1/2 hour. No time measures were taken. Thus, we do not know the duration of B's involvement in each distinct event. We followed the shadowing procedure regardless of the nature of B's work. Sometimes that meant that we observed B reading documentation only. Other times B was engaged in a wide variety of tasks.

As a general note, there is probably some self-selection of activities involved in B's choices of things to do. For instance, it is highly unlikely that B would have chosen to respond to personal email when he was being shadowed by us. As a rule, he was always directly involved in work activities. We do not consider this to be too much of a problem, however, because our goal is, after all, to build tools that help SEs work.

3.2.2 Results

The shadowing events were categorized into 14 distinct categories which are described in Table 2. Each of B's events was then classified as belonging to one of these event categories.

First, Figure 1 shows the percentage of days (for a 14 day span) on which an event occurred at least once. For example, if B searched for information one day, the search count would be incremented by 1, regardless of whether B searched 1 time, 4 times, or 24 times on that particular day.

Searching and interacting with the hardware were the most likely events to occur on a daily basis, each occurring on 8 of the 14 days. B looked at the source code on 6 of the 8 days. The reason that B searched on more days than he looked at the source code is because searching was an activity that also occurred when interacting with the hardware and debugging. B only looked at documentation on 2 of the 14 days. This is surprising because, at the time, B was still a relative novice to the software system and it is commonly assumed that novices will spend much of their time reading the documentation to get a handle on what they are doing. The data show that this was not the strategy B pursued. However, because B was a novice, it was not surprising to find that , editing code, compiling, and management were each only done on 1 of the 14 days.

The data were then examined in two distinct ways.

Figure 2 shows the proportion of each event type out of the total of 156 distinct events. Unlike Figure 1, Figure 2 shows the total count, so that if B searched 8 times on one day, that is counted as 8 instances of search.

Again, we see that overall B searched more often than he did anything else (37 times). He also frequently looked at the source code (33 times). While B was likely on any particular day to work with the hardware (see Figure 1), he did so on only 22 distinct occasions.

Remember, however, that these data do not include time measurements, but simply activity switches. So, for instance, while B did management activities on only 1 day, the code review that was undertaken took the entire 1/2 hour.

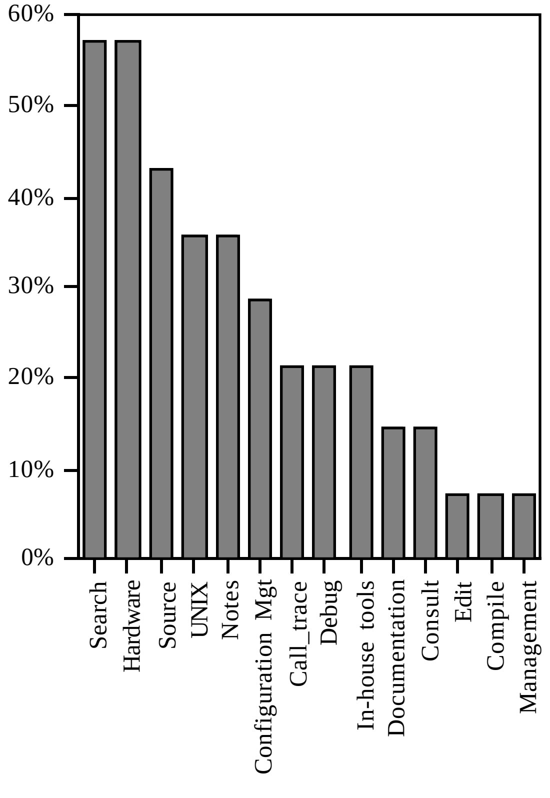


Figure 1. Percentage of days on which B engaged at least once in a particular activity.

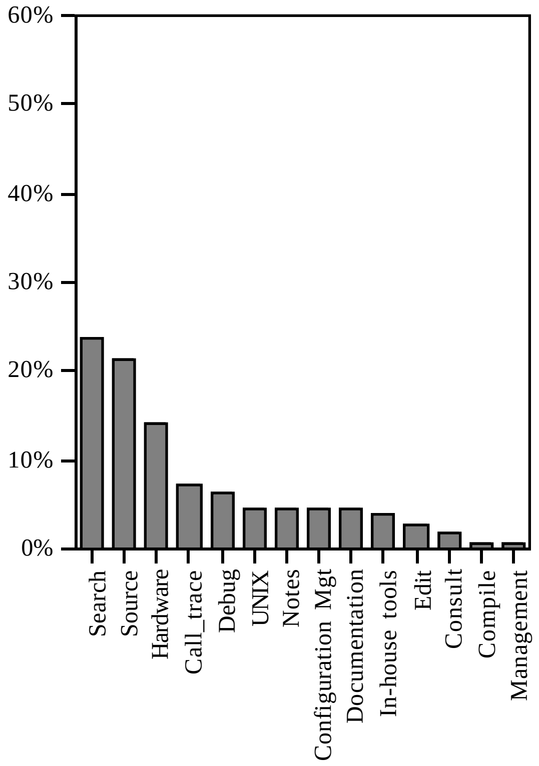


Figure 2. Percentage of times each type of event occurred out of a total of 156 distinct events.

Thus overall, in terms of both daily activities and frequency of different activities, search for information about the system, whether through Grep, in-house search tools, or within a particular editor or debugger, figures most prominently. A significant amount of effort was also expended interacting with the hardware and looking at the source code.

3.3 Group study

To generalize our findings, we have conducted several studies that focus on different aspects of the work of an entire group of SEs.

We have collected four types of data from the group. First, we asked the SEs to draw a diagram or picture of their current understanding of the system, a conceptual map, if you will. Second, we conducted intensive interviews with the SEs as they solved a real problem with the software. This generally involved 1 hour interviews over the course of several days. Third, we asked the SEs to recount how they solved a recently encountered problem. Finally, we spent one hour shadowing each SE as they went about their work. This report focuses on this fourth type of data; the shadowing data.

3.3.1 Method

3.3.1.1 Subjects

Eight group members participated in the shadowing study. Their experience ranged from the most expert member of the group (8 years) to the least experienced (6 months - recent college graduate). All but one of the shadowed subjects worked on the main controller of the hardware. One of the subjects worked primarily on the database component.

The subjects were expert in a wide variety of platforms and languages, and had experience in both development and maintenance environments.

3.3.1.2 Procedure and Data

The shadowing occurred in the same manner as for B: we sat behind the SEs and recorded the activities they engaged in. Again, a new activity was

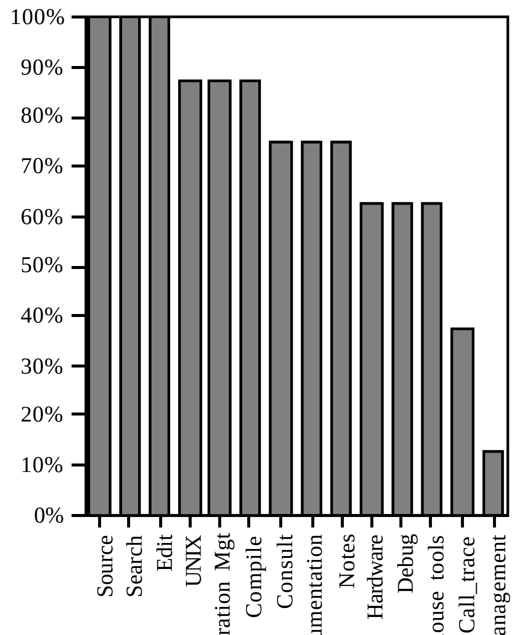


Figure 3. Percentage of users who engaged in a particular type of activity.

recorded when there was a switch in activity, so 9 Greps in a row counted as one instance of the activity search. Durations of activities were not recorded.

We recorded activities in gross, not fine, detail; e.g., we did not record the arguments to particular commands.

Shadowing schedules were not chosen to reflect any particular activity, but rather were scheduled at times convenient for the SEs. Shadowed times were relatively free from stress, i.e., SEs were not shadowed as deadlines approached.

Again, there is probably some self-selection involved in the activities that the SEs pursued. However, it was very clear that they were all working on ‘real’ problems as evidenced by their concern with the problem report’s contents.

3.3.2 Results

Like B’s data, the shadowed events were categorized into 14 distinct categories which are described in Table 2. Each of the events was then classified as belonging to one of these event categories. 356 distinct events were recorded.

Figure 3 shows the proportion of users who engaged in a particular type of activity at least once during the shadowed hour. All 8 SEs looked at the source, conducted a search, and changed the source code at least once during the hour. Most of the SEs also engaged at least once in several other activities, with 5 of the 8 SEs interacting with the hardware, debugger, or the in-house tools. On the other hand, only 3 SEs looked at a call trace, while only one SE performed a management activity.

Figure 4 shows the percentage of times a particular type of event occurred out of the total of 357 events (totaled over the 8 SEs). Issuing a UNIX command was the most frequent activity, occurring 54 times. A close second was looking at the source which was done 52 times. Interacting with the hardware or the debugger, searching, or changing the source code was done on 36, 32, 31, and 30 occasions respectively. Configuration management, consulting, compiling, and looking at in-house tools were each done about 20 times.

Surprisingly enough, reading the documentation, although done by 6 of the 8 SEs accounted for only 12 separate events. Clearly, the act of looking at the documentation is more salient in

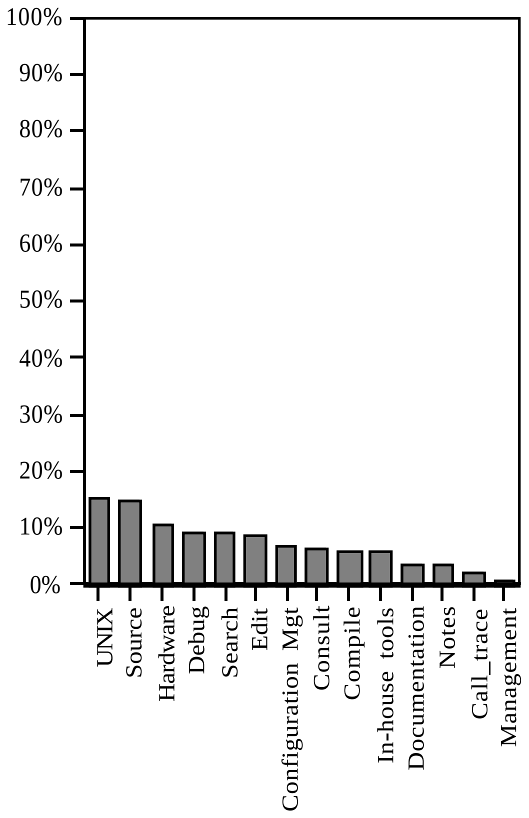


Figure 4. Proportion of times a particular type of event occurred out of the total of 357 events.

the SEs' minds (as evidenced by the questionnaire data) than its actual occurrence would warrant.

SEs only occasionally wrote notes, looked at the call trace or did management activities. This is not to say that these events are not important, but merely that they did not occur as frequently as other events.

As B did, the group frequently examined the source code. Every SE in the group made at least one search during their shadowing session, but search was less prominent than in B's activities. Search ranked as the most frequent event type for B, while it was the 4th most frequent for the group.

Code editing and compiling were more prominent activities in the group data. This is probably because B was still learning the system at the time we shadowed him, so he was not yet in a position to make many changes. This may also explain the higher prominence of call trace in his data: call trace may be effective in gaining an initial understanding of a system.

Interestingly, in-house tools and documentation were both relatively infrequent activities for both the group and B.

The group data converge with B's data to suggest that looking and searching through the source code are prominent activities for SEs. Editing and compiling also seem important. This concurs with what we would expect in that the code is the focus of their work.

3.4 Company Study

The final study we report concerns company-wide tool usage statistics. These data were obtained from the company's tool group. This group is responsible for acquiring, updating, and maintaining the company's tools. Collecting usage statistics is part of their mission.

3.4.1 Results

The data presented here represent one week of Sun tool usage by 367 users in late May. Note that this week occurred before 'vacation season,' so is fairly representative of peak tool usage. There were 79,295 separate tool calls logged from the Sun operating system. Each call counts as one usage event. These tool calls were classified according to the scheme presented in Table 3.

Figure 5 shows the proportion of times that each type of tool was used. Compilers, which accounted for 32,422 calls, or 41% of all calls are not included in this graph. This is because the compiler data include all the automatic software builds done nightly and by the various testing and verification groups. These data are therefore not representative of the SEs real work practices.

The overwhelming finding from the company data is that search is done far more often than any other activity. In fact, search accounts for 21,146 events over the course of the week, or an average of about 58 searches per individual user. Compression and un-compression tools are also used often. We never actually observed anyone using these tools. Perhaps they are used by the verification groups.

Tool	Description
Compilers	Compilers, assemblers, linkers
Compression	Compression tools such as zip and unzip
Configuration Mgt	Make and an in-house configuration management tool
Debuggers	General and in-house debuggers
Editors	Emacs, VI, and various others
Formatters	Tools such as latex and groff
Graphics Tools	Tools to create and display graphics
Hardware Connectors	In-house tools to connect to hardware
Internet Tools	Web browsers, news readers, and email programs
In-house Tools	Primarily software static analysis tools
Operating System	Windowing, terminal, and various other OS tools
Search	Primarily variations of Grep, but some in-house tools
Viewers	Document viewers such as More and Less
Other	A collection of various other tools

Table 3. Classification of the types of Sun tools.

The configuration management system was activated 2819 times, accounting for approximately 4% of all events. At this company, the configuration management system is central to the work process, both for retrieving files, filing changes, and searching through past changes (along with

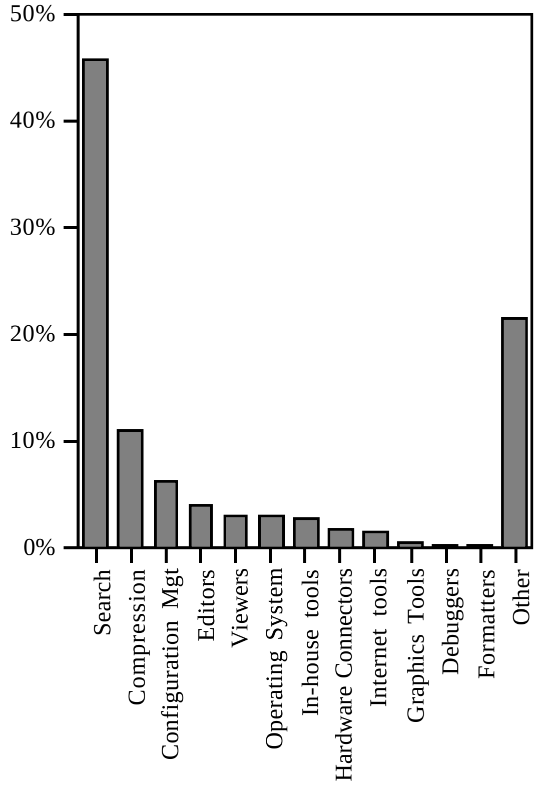


Figure 5. Proportion of all tool calls accounted for by each tool type.

associated documentation).

Editors and viewers account for approximately 3190 events, or 4% of the total number of events. This low frequency could be due to counting particularities that apply only to editors. In the company tool data, an editor command is counted only when the editor is opened. Once an editor is open, it generally stays open, regardless of how many changes are made, or how many files are viewed. In contrast, in the shadowing data, an edit was recorded each individual time the source was changed, and a source event was counted each time the source was examined, whether the editor was already open or not. Consequently, it comes as no surprise that the shadowing data edit and source frequency is higher than that of the company data.

Again, the in-house tools are not used very frequently, but that belies their importance. These tools are important because they perform necessary functions that cannot be performed by other tools.

Search is the most frequently used tool at the company wide level. Grep and its variants are the most frequently used search tools, accounting for 21,117 separate invocations. Clearly, search is an important aspect of SEs work practices.

3.5 Discussion

This examination of work practices suggests that search is an important component of real, day-to-day, software engineering. It is therefore quite reasonable to think that an improvement in search tools would help SEs to do their job better. In fact, the KBRE group has decided to focus its efforts in this direction. Currently, we are implementing a source code exploration tool [6] and investigating ways to introduce it into the workplace.

In order to improve these new tools, we are continuing our study of SE work practices in several ways. First, we are examining the source code search activity: identifying the kinds of things SEs search for, how many searches they need to find a particular piece of information, etc. Second, we are continuing our longitudinal study of B's work and our involvement with the group. Finally, we are talking to SEs at other companies to determine whether our findings generalize to their work practices.

Our shadowing studies indicate that SEs also expend a significant amount of effort in just looking at the source. This suggests that intelligent viewers might prove valuable. Indeed, the process of reading and navigating huge pieces of source code can be considered to be a type of navigation and information retrieval problem [16]. In the future, we plan on exploiting this perspective on code viewing, especially in terms of the relationship between viewers and search.

4. Application or Work-Practices Studies to the Development of Tool Requirements

We have used the data gathered during the work-practices studies described in section 3, in order to develop requirements for software engineering tools. This section describes those requirements.

4.1 The Software Engineering Task We Address: Just in Time Comprehension of Programs

Almost all the SEs we have studied spend a considerable proportion of their total working time in the task of trying to understand source code prior to making changes. We call the approach they use *Just in Time Comprehension (JITC)* [14]; the reason for this label will be explained below. We choose to focus our research on this task since it seems to be particularly important, yet lacking in sufficient tool support.

The 'changes' mentioned in the last paragraph may be either fixes to defects or the addition of features: The type of change appears to be of little importance from the perspective of the approach the SEs use. In either case the SE has to explore the system with the goal of determining where modifications are to be made.

A second factor that seems to make relatively little difference to the way the task is performed is *class of user*: Two major classes of users perform this task: Novices and experts. Novices are not familiar with the system and must learn it at both the conceptual and detailed level; experts know the system well, and may have even written it, but are still not able to maintain a complete-enough mental model of the details. The main differences between novice and expert SEs are that novices are less focused: They will not have a clear idea about which items in the source code to start searching, and will spend more time studying things that are, in fact, not relevant to the problem. It appears that novices are less focused merely because they do not have enough knowledge about what to look at; they rarely set out to deliberately learn about aspects of the system that do not bear on the current problem. The vision of a novice trying to 'learn all about the system', therefore seems to be a mirage.

As described in section 3, we observe that SEs repeatedly *search* for items of interest in the source code, and *navigate* the relationships among items they have found. SEs rarely seek to understand any part of the system in its entirety; they are content to understand just enough to make the change required, and to confirm to themselves that their proposed change is correct (impact analysis). After working on a particular area of the system, they will rapidly forget details when they move to some other part of the system; they will thus re-explore each part of the system when they next encounter it. This is why we call the general approach, just-in-time comprehension (JITC*)*. Almost all the SEs we have studied confirm that JITC accurately describes their work paradigm – the only exceptions were those who did not, in fact, work with source code (e.g. requirements analysts).

4.2 List of Key Requirements for a Software Exploration Tool

As a result of our work-practices studies (section 3), we have developed a set of requirements for a tool that will support the just-in-time comprehension approach presented in the last section. Requirements of relevance to this paper are listed and explained in the paragraphs below. Actual requirements are in italics; explanations follow in plain text.

The reader should note that there are many other requirements for the system whose discussion is beyond the scope of this paper. The following are examples:

- Requirements regarding interaction with configuration management environments and other external systems.
- Requirements regarding links to sources of information other than source code, such as documentation.
- Detailed requirements about usability.

Functional requirements. The system shall:

F1 Provide search capabilities such that the user can search for, by exact name or by way of regular expression pattern-matching, any named item or group of named items that are semantically significant[3] in the source code.

The SEs we have studied do this with high frequency. In the case of a file whose name they know, they can of course use the operating system to retrieve it. However, for definitions (of routines, variables etc.) embedded in files, they use some form of search tool (see section 4.3).

F2 Provide capabilities to display all relevant attributes of the items retrieved in requirement F1, and all relationships among the items.

We have observed SEs spending considerable time looking for information about such things as the routine call hierarchy, file inclusion hierarchy, and use and definitions of variables etc. Sometimes they do this by visually scanning source code, other times they use tools discussed in section 4.3. Often they are not able to do it at all, are not willing to invest the time to do it, or obtain only partially accurate results.

F3 Provide capabilities to keep track of separate searches and problem-solving sessions, and allow the navigation of a persistent history.

This requirement has come about because we observe users working on multiple problems and subproblems over a span of many days. We also observe them losing information they had previously found and redoing searches.

Non-functional requirements. The system will:

NF1 Be able to automatically process a body of source code of very large size, i.e. consisting of at least several million lines of code.

As we are concerned with systems that are to be used by real industrial SEs, an engineer should be able to pick any software system and use the tool to explore it.

NF2 Respond to most queries without perceptible delay.

This is one of the hardest requirements to fulfill, but also one of the most important. In our observations, SEs waste substantial time waiting for tools to retrieve the results of source code queries. Such delays also interrupt their thought patterns.

NF3 Process source code in a variety of programming languages.

The SEs that we have studied use at least two languages – a tool is of much less use if it can only work with a single language. We also want to validate our tools in a wide va-

[3] We use the term *semantically significant* so as to exclude the necessity for the tool to be required to retrieve 'hits' on arbitrary sequences of characters in the source code text. For example, the character sequence 'e u' occurs near the beginning of this footnote, but we wouldn't expect an information retrieval system to index such sequences; it would only have to retrieve hits on words. In software the semantically significant names are filenames, routine names, variable names etc. Semantically significant associations include such things as routine calls and file inclusion.

riety of software engineering environments, and hence must be prepared for whatever languages are being used.

NF4 Wherever possible, be able to interoperate with other software engineering tools.

We want to be able to connect our tools to those of other researchers, and to other tools that SEs are already using.

NF5 Permit the independent development of user interfaces (clients).

We want to perform separate and independent research into user interfaces for such tools. This paper addresses only the overall architecture and server aspects, not the user interfaces.

NF6 Be well integrated and incorporate al l frequently-used facilities and advantages of tools that SEs already commonly use.

It is important for acceptance of a tool that it neither represent a step backwards, nor require work-arounds such as switching to alternative tools for frequent tasks. In a survey of 26 SEs [7], the most frequent complaint about tools (23%) was that they are not integrated and/or are incompatible with each other; the second most common complaint was missing features (15%). In section 4.3 we discuss some tools the SEs already use for the program comprehension task.

NF7 Present the user with complete information, in a manner that facilitates the JITC task.

Some information in software might be described as ‘latent’. In other words, the software engineer might not see it unless it is pointed out. Examples of such information are the effects of conditional compilation and macros.

Acceptable limitations:

L1 The server component of the tool may be limited to run on only one particular platform.

This simplifies implementation decisions without unduly restricting SEs.

L2 The system is not required, at the present time, to handle object oriented source code.

We are restricting our focus to SEs working on large bodies of legacy code that happens to be written in non-object-oriented languages. Clearly, this decision must be subsequently lifted for the tool to become universally useful.

L3 The system is not required, at present, to deal with dynamic information, i.e. information about what occurs at run time.

This is the purview of debuggers, and dynamic analysis tools. Although it would be useful to integrate these, it is not currently a requirement. We have observed software engineers spending considerable time on dynamic analysis (tracing, stepping etc.), but they consume more time performing static code exploration.

4.3 Why Other Tools are Not Able to Meet these Requirements

There are several types of tools used by SEs to perform the code exploration task described in section 4.1 This section explains why, in general, they do not fulfill our requirements:

Grep: Our studies described in section 3.4 indicated that over 25% of all command executions were of one of the members of the Grep family (Grep, Egrep, Fgrep, Agrep and Zgrep). Interviews show that it is the most widely used software engineering tool. Our observations as well as interviews show that Grep is used for just-in time comprehension. If SEs have no other tools, it is the key enabler of JITC; in other situations it provides a fall-back position when other tools are missing functionality.

However, Grep has several weaknesses with regard to the requirements we identified in the last section:

- It works with arbitrary strings in text, not semantic items (requirement F1) such as routines, variables etc.
- SEs must spend considerable time performing repeated Greps to trace relationships (requirement F2); and Grep does not help them organize the presentation of these relationships.

- Over a large body of source code Grep can take a large amount of time (requirements NF1 and NF2).

Search and browsing facilities within editors: All editors have some capability to search within a file. However, as with Grep they rarely work with semantic information. Advanced editors such as Emacs (used by 68% of a total of 127 users of text-editing tools in our study) have some basic abilities to search for semantic items such as the starts of procedures, but these facilities are by no means complete.

Browsing facilities in integrated development environments: Many compilers now come with limited tools for browsing, but as with editors these do not normally allow browsing of the full spectrum of semantic items. Smalltalk browsers have for years been an exception to this, however such browsers typically do no not meet requirements such as speed (NF2), interoperability (NF4), and multiple languages (NF3). IBM’s VisualAge tools are to some extent dealing with the latter problem.

Special-purpose static analysis tools: We observed SEs using a variety of tools that allow them to extract such information as definitions of variables and the routine call hierarchy. The biggest problems with these tools were that they were not integrated (requirement NF6) and were slow (NF2)

Commercial browsing tools: There are several commercial tools whose specific purpose is to meet requirements similar to ours. A particularly good example is Sniff+ from Take5 Corporation [17]. Sniff+ fulfills the functional requirements, and key non-functional requirements such as size [NF1], speed [NF2], and multiple languages [NF3]; however its commercial nature means that it is hard to extend and integrate with other tools.

Program understanding tools: University researchers have produced several tools specially designed for program understanding. Examples are Rigi [11] and the Software Bookshelf [5]. Rigi meets many of the requirements, but is not as fast [NF2] nor as easy to integrate other tools [NF6] as we would like. As we will see later it differs from what we would like in some of the details of items and relationships. The Software Bookshelf differs from our requirements in a key way: Before somebody can use a ‘bookshelf’ that describes a body of code, some SE must organize it in advance. It thus does conform fully with the ‘automatically’ aspect of requirement NF1.

4.4 The Tools We are Developing

As a consequence of our work practices studies, and thus the requirements described in the last section, we have developed an improved software exploration tool which we call tksee. A view of this tool is shown in figure 6.

The main features that fulfill F1 and F2 (search capabilities) are in the bottom two panes. The bottom left pane shows a hierarchy that the user incrementally expands by asking to show attributes of items, or to search for information (relations or Grep results) about a given item. The currently selected item is shown in the bottom-right pane, from which the user can hyper-jump by selecting any item of text.

The main feature that fulfills F3 is the top pane. Each element in this pane is a complete state of the bottom two panes. A hierarchy of these states is saved persistently, so each time the user starts the tool, his or her work is in the same state as at the end of the previous session.

The non-functional requirements are met by the tksee architecture shown in figure 7. This architecture includes a very fast database, an interchange language for language-independent information about software, and a client-server mechanism that allows incorporation of existing tools (e.g. Grep) so software engineers can continue to use tools they already find useful.

Further details about this tool are in [6, 15].

We are continuing our involvement with users: we are studying how their work practices evolve when they choose to adopt this tool.

5. Conclusions

In conclusion, the study of work practices provides a path to tool design that is an alternative to the traditional paths taken in human-computer interaction, namely those issuing from the study of the users’ cognitive processes and mental models, and the emphasis on usability. The problem of disuse has plagued software tools designed with these traditional human computer interaction approaches. By focusing on workplace activities, the study of work practices increases the likelihood

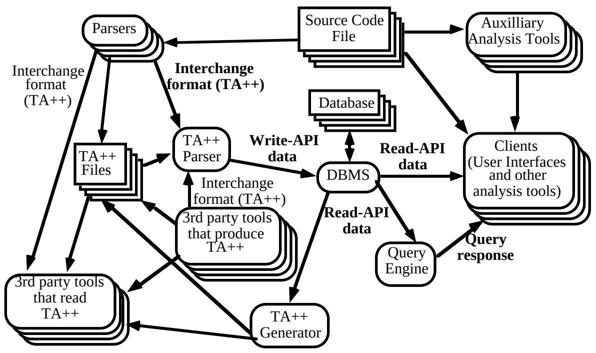
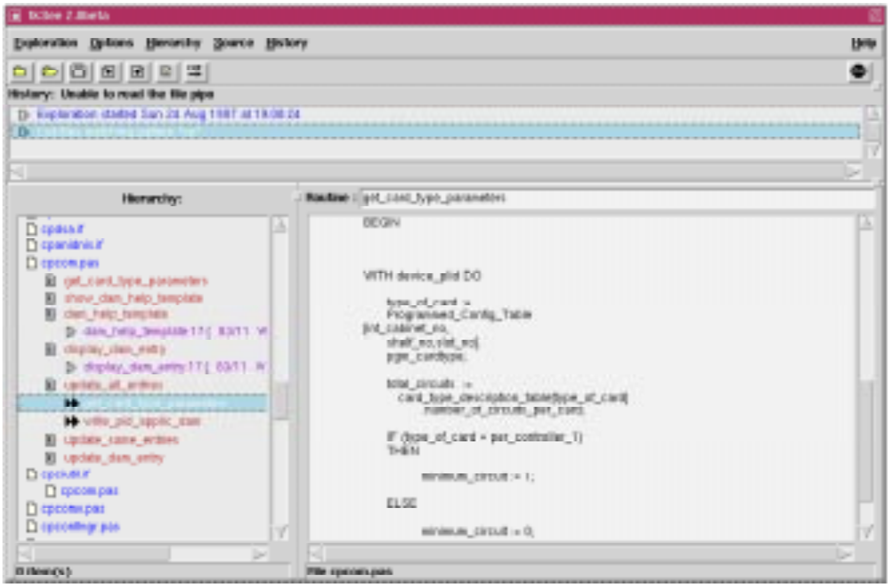

Parsers
Source Code File
Auxilliary Analysis Tools
Interchange format (TA++)
Interchange format (TA++)
Database
TA++ Files
TA++ Parser
Write-API data
Read-API data
Clients (User Interfaces and other analysis tools)
DBMS
Interchange format (TA++)
3rd party tools that produce TA++
Read-API data
Query Engine
Query response
3rd party tools that read TA++
TA++ Generator

Figure 7: Data flow diagram showing architecture of the tksee software

that tools can be smoothly integrated into the users' daily activities. This, in turn, should increase the acceptance and use of software tools designed on the basis of work practices. Whether one wishes to examine user cognitions or not, it is necessary that tools be consistent with work practices for them to be used. Once this consistency is established, the usability approach can be taken to ensure that the SEs can effectively use these tools to accomplish their work.

It is possible that the study of work practices can reduce, or perhaps even eliminate, the need to study cognitive processes and mental models. This will depend on the accuracy and detail with which work practices can be described. If they can be described in detail, in terms of every system state explicitly and intentionally accessed by the user, it may not be necessary at all to fathom the users' cognitions. We may only need to abide by general principles of usability and usability testing in addition to the work practice specifications in order to design useful, and used tools. Moreover, it may be more efficient, in terms of time, to take the work practice approach to tool design than the cognitive approach. However, further empirical work is required in order to strengthen out confidence in these statements. Further details about our research can be found in [15].



```
BEGIN


WITH device_pid DO

    type_of_card :=
    Programmed_Config_Table
[pid_cabinet_no,
    shelf_no,slot_no]
    pge_cardtype;

    total_circuits :=
      card_type_description_table[type_of_card]
        number_of_circuits_per_card;

    IF (type_of_card = pat_controller_1)
    THEN

          minimum_circuit := 1;

    ELSE

          minimum_circuit := 0;
```

Figure 6: The main window of the tksee tool.

Acknowledgments

We would like to thank the SEs who participated in our studies, in particular those with whom we have worked for many months. We would also like to thank the tools group at the company for providing us with the tool usage statistics. Finally, we would like to thank the KBRE group for many helpful suggestions.

About the Authors

Janice Singer is a cognitive psychologist who is now researching software engineering work practices with the Software Engineering Laboratory at the National Research Council of Canada. Prior to her Ph.D. studies, she conducted research in human-computer interaction and worked as a software engineer.

Timothy C. Lethbridge is an Assistant Professor in SITE at the University of Ottawa. He teaches software engineering and human-computer interaction and heads the Knowledge-Based Reverse Engineering group, sponsored by CSER. Dr. Lethbridge has worked as a software developer in both public and private sectors.

Norman Vinson is a cognitive psychologist working in the Interactions with Modeled Environments group, at the Institute for Information Technology, National Research Council of Canada. Prior to joining the NRC, Dr.

Vinson was a user-interface designer at Northern Telecom.

Nicolas Anquetil recently completed his Ph.D. at the l’Université de Montréal and is now working as a research associate and part time professor in SITE at the University of Ottawa.

The authors can be reached at {singer, vinson}@iit.nrc.ca and {tcl, anquetil} @site.uottawa.ca The web site of the KBRE project is www.csi.uottawa.ca/~tcl/kbre, and that of the Institute for Information Technology is www.iit.nrc.ca.

References

[1] Anderson, J., *Cognitive Psychology and Its Implications*, WH Freeman, 1995..

[2] Blomberg, J., Suchman, L., & Trigg, R., Reflections on a Work-oriented Design Project. *Human Computer Interaction (11)*, pp. 237-265, 1996.

[3] Beyer, H., & Holtzblatt, K., Apprenticing with the customer. *CACM (38*), pp. 45-52, 1995.

[4] Brooks, R., Towards a Theory of the Comprehension of Computer Programs, *Int. J. of Man-Machine Studies (18)*, pp. 543-554, 1983.

[5] Holt, R., Software Bookshelf: Overview And Construction, www.turing.toronto.edu/~holt/papers/bsbuild.html

[6] Lethbridge, T., & Anquetil, N., Architecture of a source code exploration tool: A software engineering case study. SITE, Technical Report.

[7] Lethbridge, T. and Singer J., Understanding Software Maintenance Tools: Some Empirical Research, *Workshop on Empirical Studies of Software Maintenance (WESS 97)*, Bari Italy, October, 1997.

[8] Lethbridge, T. and Singer, J, Strategies for Studying Maintenance", *Workshop on Empirical Studies of Software Maintenance*, Monterey, November 1996.

[9] Littman, D., Pinto, J., Letovsky, S., & Soloway, E., Mental Models and Software Maintenance, *Empirical Studies of Programmers*, pp. 80-98, 1986.

[10] Mayhew, D., *Principles and Guidelines in Software User Interface Design*, Prentice Hall, 1991.

[11] Müller, H., Mehmet, O., Tilley, S., and Uhl, J., A Reverse Engineering Approach to Subsystem Identification, *Software Maintenance and Practice*, Vol 5, 181-204, 1993.

[12] Pennington, N., Stimulus Structures and Mental Representations in expert comprehension of computer programs. *Cognitive Psychology (19)*, pp. 295-341, 1987.

[13] Singer, J. and Lethbridge, T, Methods for Studying Maintenance Activities, *Workshop on Empirical Studies of Software Maintenance*, Monterey, November 1996.

[14] Singer, J., and Lethbridge, T. (in preparation). Just-in-Time Comprehension: A New Model of Program Understanding.

[15] Singer, J, Lethbridge, T., and Vinson, N. Work Practices as an Alternative Method for Tool Design in Software Engineering, *CHI ‘98.*

[16] Storey, M., Fracchia, F., & Müller, H., Cognitive Elements to support the construction of a mental model during software visualization. Proc *5th Workshop on Program Comprehension*, Dearborn, MI, pp. 17-28, May, 1997.

[17] Take5 Corporation home page, http://www.takefive.com/index.htm

[18] Vicente, K and Pejtersen, A. *Cognitive Work Analysis*, in press

[19] von Mayrhauser, A., & Vans, A., From Program Comprehension to Tool Requirements for an Industrial Environment, In: *Proc. 2nd Workshop on Program Comprehension*, Capri, Italy, pp. 78-86, July 1993.

[20] von Mayrhauser, A., & Vans, A., From Code Understanding Needs to Reverse Engineering Tool Capabilities, *Proc. 6th Int. Wkshp on Computer-Aided Software Engineering,* Singapore, pp. 230-239, July 1993.

[21] von Mayrhauser, A and & Vans, A., Program Comprehension During Software Maintenance and Evolution, *Computer*, pp. 44-55, Aug. 1995.