3
Studies of the Work Practices of Software Engineers

Timothy Lethbridge
Janice Singer

3.1 Introduction

In this chapter we describe various techniques for studying and representing the work of software engineers[1] (SEs) and using the results to develop requirements for software engineering tools.

The ultimate objective of our research is to discover techniques that will enable software engineers to more productively make changes to large legacy real-time software systems. However, to achieve this objective we must understand software engineers' work practices. We describe various techniques we employed to observe work practices, analyze the resulting data, and produce graphical models of work patterns. In particular, we describe techniques that we have developed such as *synchronized shadowing* and the use of Use Case Maps to represent *work patterns*. Finally, we highlight some of the results of using these techniques in a real project: An important observation is that efficiently performing searches within source code is of paramount importance to the SEs when they work with large bodies of source code.

Our work began with collaboration between a computer scientist building tools for software maintainers (T. Lethbridge) and a psychologist (J. Singer) just hired in a software engineering research group. Although our research goal was to improve software maintainers' productivity, improving productivity was a very open-ended problem. It was not at all clear, when we began our work, what aspects of software maintenance could be most improved; it was even less clear

[1] By *software engineers*, we are referring to people who perform software engineering work, but who may not be Professional Engineers in the legal sense.

what tools would be appropriate. However, since we both had a background in human/computer interaction, we were certain that we wanted to build *usable* tools for software maintainers.

Our first task involved a literature review. From the Empirical Studies of Programmers workshops, there were a few papers on software maintainers (e.g., Litman et al., 1996; Boehm-Davis et al., 1992). This research, primarily conducted from an information processing perspective, helped us understand individual processes in software maintenance. However, it was not clear how to take these results and build systems that could be used in real industrial practice with real software engineers. As Curtis (1986) appropriately asked about these types of studies, "By the way, did anyone study any real programmers?" meaning that the results might not apply in industrial practice.

There have been field studies in software design (Walz et al., 1993; Curtis et al., 1988; Kraut and Streeter, 1995), but again, it is not clear how design relates to maintenance. Also, these studies tended to look at larger issues that were not necessarily pertinent to building tools for individual software engineers.

Bendifallah and Scacchi (1987) did look at software maintenance as a form of articulation work. However, again, their work examines academic researchers and their maintenance of a relatively small tool. While Bendifallah and Scacchi's work assured us that the study of work practices was feasible in this field, it was not clear how their results would generalize to our target group of industrial maintainers.

This lack of relevant literature led us to broaden our emphasis from *usability* to *usefulness.* The questions of what do software maintainers do on a daily basis, in what activities are they involved, with what frequency, and using which tools, were all unanswered in the literature. Without this knowledge, we could not be sure about what would be useful tools for this domain. Thus, we went back to the literature. A review of the work of researchers in participatory design (e.g., Kyng and Mathiassen, 1997), distributed cognition (e.g., Hutchins, 1994), situated cognition (e.g., Suchman, 1987), and activity theory (e.g., Bannon and Bødker, (1991)[2] led us to believe that we could not ignore the context within which the work took place. Because of this, we decided to implement a field study in software maintenance (Singer and Lethbridge, 1998a; Lethbridge et al., 1997). This study used several ethnographic methods for data collection, including questionnaires, interviews, and observations. Following Glaser and Strauss' (1967) grounded theory approach, we were able to determine that soft-

[2] A comprehensive review of this literature will not be undertaken in this paper. Please refer to the individual papers and/or books for further information.

ware maintainers typically follow a *just-in-time comprehension* approach to program comprehension. That is, at any given time, they understood only the specific portion of the source code that would help them solve their current problem.

While we were closer to an answer about the usefulness of different software maintenance tools, we were unhappy with our methods of data collection and analysis. The difficulty of moving from field work to design requirements has been highlighted by other researchers (e.g., Button and Dourish, 1996; Blomberg et al., 1996; Simonsen and Kensing, 1997)[3]. In fact, tools are now being created to help researchers record and represent their understanding of work (Pycock et al., 1998; Jordan et al., 1995).

Our dissatisfaction, however, focused not on adequate representations of the field, but rather on the fact that we felt we were collecting the wrong data. The field data was incomplete because software engineers were so quick that it was impossible to get all the actions recorded in observation sessions. Technical and practical issues did not allow us to videotape sessions. Additionally, our observations did not allow us to answer certain fundamental questions such as what is an individual's goal in doing a task, or how much time is spent on a task. Finally, the work required to represent the data was extreme. We spent over six months making transcripts and poring over them. While this might be appropriate in a research environment, it is entirely unfeasible in industry. These two concerns led us to develop both a new method for collecting data, *Synchronized Shadowing*, and a new method for representing it, using Use Case Maps (UCMs).

This chapter will discuss both of these innovations in the context of another field study that we subsequently implemented. We begin with a discussion of a general model of empirical studies in software engineering and situate our own observational field studies within this. Then we give more details about Synchronized Shadowing and our use of UCMs. We conclude with a case study showing how we applied these techniques to meet our objectives of building effective tools.

The methodology we developed as a result of our work is outlined in Figure 3.1. It combines synchronized shadowing with the use of Use Case Maps. We call it Work Analysis with Synchronized Shadowing (WASS).

[3] Again, these references are in no way meant to be a comprehensive view, but rather an overview of current thinking.

3.2 An Overview of Approaches to Empirical Studies of Software Engineering Practices

Broadly defined, one calls a study *empirical* if it involves observing or measuring something. In analytical studies, in contrast, one deduces conclusions by

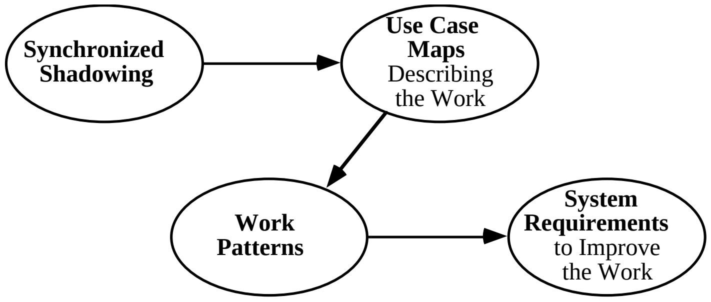

Synchronized Shadowing
Use Case Maps Describing the Work
Work Patterns
System Requirements to Improve the Work

Figure 3.1. The Work Analysis with Synchronized Shadowing (WASS) Methodology.

applying logical and other mathematical reasoning to physical laws and other established facts. Since there are few unchallengable facts in the domain of software engineering processes, research in this domain will normally be empirical in nature.

Figure 3.2 shows a way of categorizing empirical studies of software processes using three dimensions: The environment, the degree of human contact, and the level of control. Most of the research discussed in this chapter falls close to the origin; it involves interactively gathering information about what people do in their natural work environments: We refer to this as *work practices* studies.

3.2.1 Natural vs. Artificial Environments

The first dimension in Figure 3.2 distinguishes between field studies in natural work environments and studies performed in laboratory environments.

Field studies are conducted with practicing software engineers in the industry, whereas laboratory studies often involve groups of students in classroom or lab settings. Field studies often take more effort than laboratory studies. Relationships must be established with industrial companies, suitable software projects and individual participants must be found, and the uncertain nature of the day-to-day activities of the company and its employees mean that the direction of the research is somewhat out of the researchers' hands (Lethbridge et al., 2000).

For the most part, studies of students performing software engineering tasks in laboratories are easier to conduct since there is a ready supply of students in university classes, and a faculty member can dictate their goals. While the conclusions of laboratory studies are useful, they are not as likely to be relevant to industrial practice since students lack experience and goals, and since their methods will not normally be the same as those of industrial practitioners.

One counter example is that Porter and Votta (1998) found no difference in results for professional vs. graduate student programmers. However their experiment was artificial in the sense that the exercises used were designed strictly for the experiment. It might be that people work differently with known materials than experimental ones. It is probably also true that graduate students are closer to professionals than undergraduates are in terms of their programming ability. Nonetheless, since we wanted to absolutely ensure our work was industrially relevant, the work discussed in this chapter falls in the field study category: We studied real programmers in real industrial environments.

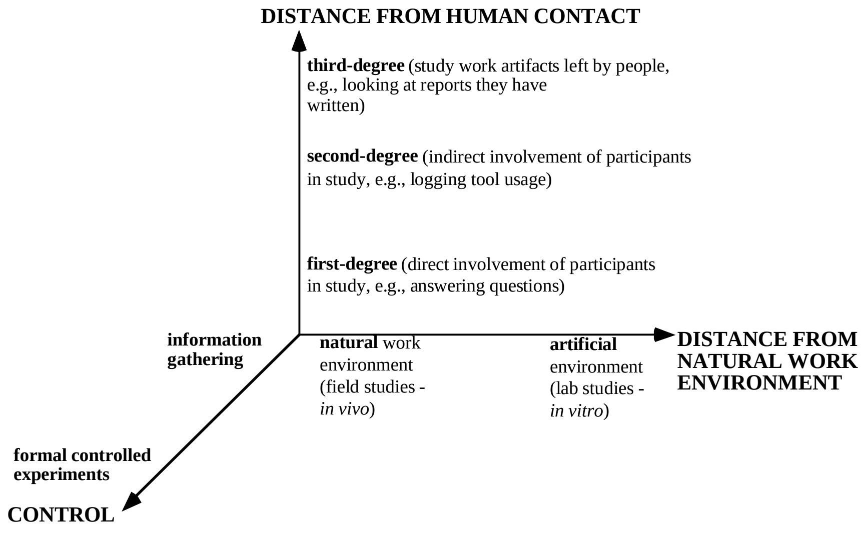

DISTANCE FROM HUMAN CONTACT
third-degree (study work artifacts left by people, e.g., looking at reports they have written)
second-degree (indirect involvement of participants in study, e.g., logging tool usage)
first-degree (direct involvement of participants in study, e.g., answering questions)
information gathering
natural work environment (field studies - in vivo)
artificial environment (lab studies - in vitro)
DISTANCE FROM NATURAL WORK ENVIRONMENT
formal controlled experiments
CONTROL

Figure 3.2. An approach to categorization of empirical studies of software processes.

3.2.2 Degree of Human Contact

The second way of categorizing empirical studies shown in Figure 3.2 relates to the degree of human contact that the technique involves.

We define *first-degree* empirical studies to be those involving human-to-human interaction between researchers and participants. Such techniques can include brainstorming, interviews, surveys, and observational studies.

We consider *second-degree* studies to be those where human processes are monitored, but where the researchers do not interact directly with the humans themselves, e.g., by gathering information automatically as people work.

Third-degree studies involve analysis of the artifacts resulting from work, e.g. source code, documents, and problem reports. The work reported in this chapter was first-degree (primarily observational studies), although we also used some second- and third-degree information.

3.2.3 Information-Gathering vs. Experimentation

The final dimension in Figure 3.2 contrasts *information-gathering studies* with *experiments*. Information gathering studies are suitable for generating hypotheses, while controlled experiments are suitable for confirming or testing hypotheses.

In general, information-gathering studies are used to gather raw information about a phenomenon; the information may then be used to build a qualitative or quantitative model of the phenomenon. Techniques for producing qualitative models are discussed in detail by Glaser and Strauss (1967). Neuman (1997) and Denzin and Lincoln (1994) also give more detailed information on how to conduct qualitative studies.

Quantitative models can be used to develop hypotheses that can be tested in experiments. Experiments require the existence of a model and hypotheses about that model that are to be tested; formal experiments follow the scientific method rigorously and involve setting up some situation where extraneous variables are controlled, varying some independent variable(s), and measuring some dependent variable(s) in order to refute the null hypothesis.

In software engineering, it is usually very difficult to adequately control the extraneous variables, so true experiments are less widely used. However, there is one type of experiment that sits in the middle of this continuum and is useful in software engineering. Quasi-experiments are experiments where subjects are not randomly assigned to treatments. For instance, one could conduct a quasiexperiment on two different groups who had decided to implement two different programming processes. Here the two groups were not assigned randomly to the processes, but rather self-selected to them. For more information on quasi-experiments, see Cook and Campbell (1979).

Our studies of work practices are information-gathering in nature since we want to describe and model the work of software engineers.

3.2.4 Summary of the Three Dimensions of Empirical Studies

The three dimensions of empirical studies are largely orthogonal, with all points in space being possible, although not equally probable. For example, experiments are more likely to be performed in artificial environments where it is easier to control variables. Nevertheless it is possible to conduct large-scale experiments in an industrial context - for example, competitive development of a product by two teams that use different tools

In the above three-dimensional model of empirical studies, we have organized the types of empirical studies according to their data-generation phase, which we discuss in more detail in the next section. However, the resulting information must also be analyzed so conclusions can be drawn. This is usually the most time-consuming phase, since vast amounts of data can be generated, especially for information-gathering studies. Techniques for the analysis phase are discussed in section 3.4.

3.3 Techniques for Gathering Data in Observation Sessions

One of the techniques most widely used to understand work practices is observation. Shadowing is a form of observation where the observer moves around with the observee, recording what they are doing as they go about their normal daily routine. There are two big difficulties with shadowing: one is to effectively *capture* information; the other is to *analyze* the copious resulting data. To capture data, there are two widely used alternatives, simple note-taking or videotaping. Both require the output to be coded following the observation session before any analysis is undertaken. In this section we review these classic techniques, and then present our synchronized shadowing technique.

3.3.1 Note-Taking and Videotaping

Simple note-taking has two key problems: First, many details may be missed by the note-taker, partly because he or she may not notice all the nuances of what is going on, and partly because it is difficult to rapidly take accurate notes using a consistent format. Second, it is not feasible for the note-taker to record precise times when events occur, especially where action occurs quickly - the process of looking at his or her watch would cause the note-taker to miss important activity.

Videotaping does not suffer from these problems since it allows one to record almost all details of a session. However, the process of coding can be very time consuming because one has much more data to work with.

There are some automatic logging tools that record precisely what occurs on the computer, such as every key press or every mouse click. These are impractical for our purposes, though, for two key reasons. First, they only record computer activity. We are interested in obtaining information about the work environment which includes situations when the participant looks at documentation, talks to neighbors, etc. The automatic logging tools do not capture this information. Second, because many programmers personalize their computer environment, the output of these tools in often difficult to interpret. For instance, the tools might tell you that a programmer is in Emacs, but they would not be able to interpret the macros that the programmer has set up to search for specific strings in Emacs. This makes these tools less useful in our context.

New tools are being developed that record the screen as well as the user’s voice and synchronize them. We have not tried these tools because they have one fundamental problem in our context: Our software engineers move around from place to place, using different computers (e.g., in special hardware labs). We do not want to use observation techniques that interfere with the natural work processes.

3.3.2 Synchronized Shadowing

In our case, to make shadowing more practical, we developed an approach that has many of the advantages of videotaping, but without many of the drawbacks of note-taking. Our approach uses a program on a laptop computer to provide automated assistance to note-takers. The program improves the note-taking process in the following two ways:

- The note-taker can simply press one of many *buttons* to record an event that recurs frequently. This results in substantially increased note-taking speed, and hence fewer missed events. The notes will also be more consistent and hence faster to analyze since the buttons correspond to event categories: Much of the coding takes place at the time of observation. The meaning of the buttons can be preassigned following pilot studies, although adding new buttons dynamically during the observation session remains a possibility. Also, the note-taker can type other information after pressing any button, so nothing is lost from the ordinary note-taking process.
- Timing information is automatically recorded along with every button press, allowing for a level of accuracy in data analysis that would normally be available only by analyzing videotape.

Automated note-taking as described above can be very useful, but we developed the technique one step further: A single person using our program will still tend to miss much information. This makes sense because it is well known that when analyzing videotapes, one has to replay sections of the tape several times in order to notice all the details. We therefore arrange for *two* note-takers to participate in the shadowing, each using the automated note-taking program, but with different meanings for the buttons so they record somewhat different aspects of the work being observed. The two records are be merged after the session to form a more complete picture of what happened. A key process that makes this merging feasible is synchronizing the clocks of the two laptop computers so that proper sequences of events can be reconstructed - for this reason, we call our approach *synchronized shadowing* (Singer and Lethbridge, 1998).

It is very simple to create basic synchronized shadowing tools. For the event-recording buttons, we created macros in Microsoft Word that redefine certain keys (control sequences or function keys). Each such button adds a time stamp as well as an identifying string of characters to the current document. Before the observation session starts, we synchronize the computers' clocks. After the session we concatenate the documents and sort them. Figure 3.3 is the source code for two of the macros and Figure 3.4 illustrates the output of a session.

```
Public Sub MAIN()
Dim count_$
Dim counti
count_$ = WordBasic.[GetDocumentVar$]("statementNumber")
counti = WordBasic.Val(count_$)
counti = counti + 1
count_$ = Str(counti)
WordBasic.SetDocumentVar "statementNumber", count_$
WordBasic.InsertPara
WordBasic.Insert "* "
WordBasic.Insert count_$
WordBasic.Insert " - "
WordBasic.InsertDateTime DateTimePic:="H:mm:ss", InsertAsField:=0
WordBasic.Insert " "
End Sub

Public Sub MAIN()
WordBasic.ToolsMacro Name:="InsDate", Run:=1
WordBasic.Insert "GREP = "
End Sub
```

Figure 3.3. Two example MS-Word macros for automated note-taking while shadowing. The first inserts time, and the second inserts the time plus the keyword GREP-it is an example of one of many macros that would be bound to specific buttons.

```
   1 13:32:40 NEW-GOAL Friday, August 01, 1997 Jane Smith
   2 13:39:26 still explaining stuff
   3 13:39:52 UNIX ls cd
   4 13:40:12 EDITOR srh
   5 13:40:22 EDITOR quit
   6 13:40:28 GREP in system
   7 13:40:40 VIS at results
   8 13:41:02 EDITOR open found file
   9 13:41:33 EDITOR open empty
   10 13:41:45 EDITOR copy
   11 13:41:52 EDITOR paste
   12 13:42:00 EDITOR save as xxdbllq.c
   13 13:42:21 MODIFY part of query text
   14 13:44:08 EDITOR save
   15 13:44:12 MODIFY func name
   16 13:44:38 EDITOR save
   17 13:44:59 stop observing
```

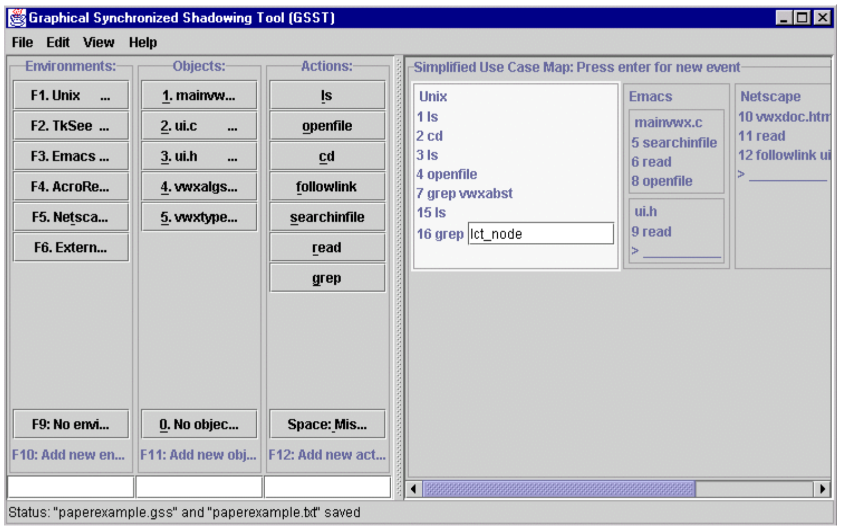
Figure 3.4. Output of a synchronized shadowing session using MS-Word macros (lower-case text was typed by the person operating the program).

In addition to MS-Word macros, we have also created a more advanced synchronized shadowing tool, illustrated in Figure 3.5. This tool allows events to be nested within two levels of activities. For example, the highest level of activity might represent the primary task being performed; the second level might be the tool being used; and the events might be specific actions performed with the tool. Our new synchronized shadowing tool allows the user to dynamically add events, and manipulate various preferences. It provides similar output to that shown in Figure 3.4.

Synchronized shadowing is not perfect: The note-takers tend to vary the amount of time between the occurrence of an event and pressing the appropriate button. Timing, therefore is likely to be accurate only to the nearest 10 or 15 seconds, but this is adequate for our purposes.

3.4 Modeling Work to Develop Requirements

In this section we discuss how we use various techniques, including Use Case Maps (UCMs), to analyze the data obtained by synchronized shadowing. In the next section we provide a case study, illustrating the use of these techniques.



Unix
1 ls
2 cd
3 ls
4 openfile
7 grep vwxabst
15 ls
16 grep lct_node

Emacs
mainvwx.c
5 searchinfile
6 read
8 openfile
ui.h
9 read
>

Netscape
10 vwxdoc.htm
11 read
12 followlink ui
>

Status: "paperexample.gss" and "paperexample.txt" saved

Figure 3.5. User interface of our second-generation Graphical Synchronized Shadowing Tool (GSST).

3.4.1 Coding Observational Data and Detecting Patterns

It is very difficult to analyze data that result from observational studies. The first step is to code, or categorize, all interesting events that occurred during the session. This is a subjective process normally requiring several iterations as the coders refine the coding scheme.

Once the raw observations are coded (at least preliminarily), there are two important approaches that can be used individually or together to obtain interesting information from the coded data:

- Counting occurrences of types events or summing the total amount of time spent on classes of activities. This can be useful to give an overall impression of how people spend their time. If one has enough data for different classes of people, one can discover differences among the classes.
- Detecting and modeling patterns of activities. This involves looking for repeating sequences and cycles that can be used to describe parts of the observed activity at a higher level of abstraction. Doing this is described below.

The above approaches can be used synergistically. The counting and analysis of occurrences of these patterns can follow the process of modeling and building patterns. Similarly, the process of counting can lead to the development of patterns by pointing out the important types of events that should be included in those patterns.

A useful first step in discovering and representing patterns of activities is finding subsequences that are repeated frequently in the coded data. Several algorithms are known that can help with this. A basic approach simply divides an entire coded sequence of events into all possible subsequences of length *n* (called n-grams where n is normally at least 3) and counts the occurrences of each n-gram. Useful subsequences appear as the n-grams that occur most frequently. Even more interesting subsequences can sometimes be found by progressively increasing the value of *n*.

Exploratory Sequential Data Analysis (ESDA), (Sanderson et al., 1994) is another well-known technique that has been applied to describing software engineering processes (D'Astous and Robillard, 2000).

We found, however, that we wanted to go beyond merely finding patterns that are sequences of events. We sought a graphical technique that could show the context of each event and could more actively assist software designers to develop tools, as has been advocated by Bannon (1994) and Suchman (1995).

3.4.2 Use Case Maps

The Use Case Map (UCM) notation shows multiple sequences of localized events. By localized, we mean that each event occurs in a particular *context*. Contexts are shown as boxes. Sequences are shown as paths that wind from context to context, may form loops and may split into independent sequences or may merge. Events are points on the paths.

The UCM notation was originally invented by Buhr (Buhr 1998; Buhr and Casselman, 1996) to represent causal flows of responsibilities in real-time software systems. In such systems there are normally several parallel processes or tasks (paths), interacting with different subsystems (contexts) and involving interactions or computations (events).UCMs are also ideal, however, to represent the detailed flow of the tasks of a single person or a small group. As with computer systems, people work in parallel on multiple tasks (paths), work with various different tools, documents or other people (contexts), and perform series of actions (events).

Figure 3.6 shows a UCM that is being used to model a user's *particular* interaction with Unix and the Emacs editor. Later in the chapter we will show additional UCMs containing *generalized* patterns crystallized from observing many users.

To understand Figure 3.6, follow the numbered points along the path, and read the descriptions below:

- The circle is the start symbol.
- The user enters Emacs and opens a directory, the listing of which is shown as the first inner box. The user enters the context of this directory.
- The user employs an item from the directory listing to initiate the opening of a file. The bold arrows indicate information being taken to be used later.
- After entering the context of a file, the user performs a search.The user performs another search; the loop indicates repetition.
- The user places some information in the copy buffer, to be used later.
- After leaving Emacs and entering the context of the Unix command line, the user issues a *grep* command, using information in the copy buffer as the argument (represented again by the bold arrow).

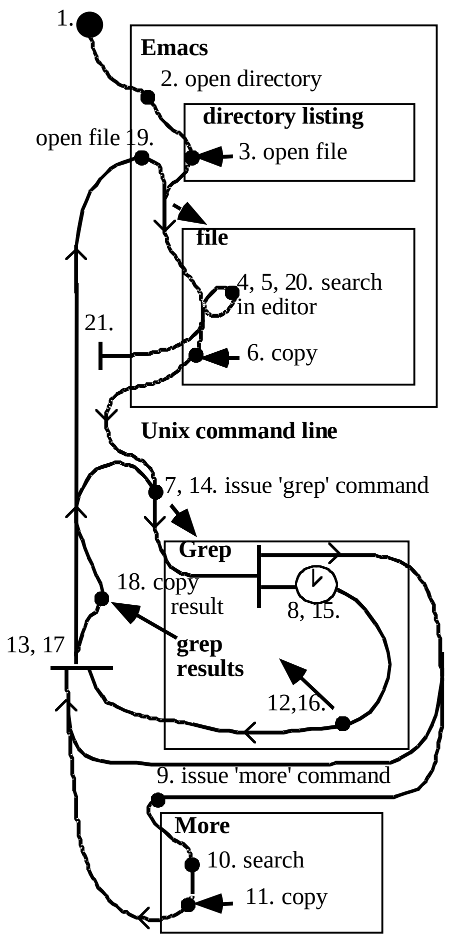


Figure 3.6. An example Use Case Map (UCM) shows the flow of work from context to context.

- In the newly created *grep* context, the path forks. The upper path is that taken by the user who wishes to do something else while *grep* is executing. The lower path, with the clock symbol, is that taken by the computer that takes time to perform the *grep* (the clock symbol indicates a delay that could at some point be cancelled if the user gets tired of waiting).
- The user reenters a Unix command-line context and issues a *more* command.
- The user searches using the *more* tool.
- The user copies some text from the information displayed by *more*.

- Meanwhile, the *grep* started earlier has completed and produced its results.
- The user now can get back to dealing with the *grep* results. The horizontal line indicates an *and-join* or *synchronization* in which the two subpaths (waiting for *grep* and the users activities while waiting) are reunited after *both* of them are completed.
- The user could do something with the *grep* results, but decides not to. Instead he ignores them and issues another *grep* command using the information copied earlier while performing *more*. The path converges with the path taken earlier using an *or-join*; the user will now repeat a subsequence performed earlier.
- There is a fork and a delay again during the execution of *grep*; the user again has the opportunity to do other things while waiting.
- As before, the results are eventually returned.
- The user waits at the and-join for the *grep* to be complete; this time he chooses not to work in the *more* program.
- This time, the user copies some text from the result.
- Instead of repeating *grep* for a third time, this time the user goes back to the Emacs context and opens a file using the contents of the copy buffer (taken earlier from the *grep* results) as a file name.
- The user searches in the file using Emacs as he did earlier.
- The user finishes his task, as shown by the line terminating the path.

The process of creating a UCM from synchronized shadowing data is relatively simple, although it is currently a manual process. Proceeding through the data sequentially, one draws path segments from event to event, drawing new contexts and placing events inside them as needed (the contexts need to be coded as part of the synchronized shadowing process). When a sequence is repeated, one makes the path form a loop (having previously detected repeated sequences as described above helps one anticipate such loops). We have found that after a small amount of rearranging, a readable UCM normally emerges.

If a UCM becomes too difficult to read, i.e., with too many events, contexts and paths, it can be split into several UCMs, each containing paths and contexts extracted from the large messy UCM. Doing this is how we discover *work patterns* in the UCMs: A work pattern is shown as a simplified UCM that contains paths that are followed very frequently, and typically involve just one or two contexts. UCMs provide a mechanism called *stubs* and *plug-ins* to facilitate this.

There are also other notations that can be used to model human work:

- data flow diagrams can, at a very high level, show the movement of information around a business;
- work-flow diagrams and Petri nets can show the sequences and dependencies among subtasks;
- and flow charts can show decision-making processes.

None of these notations, however, can clearly show at a detailed level both the context of the work and the multiple interacting threads of events. UML activity charts perhaps come closest to what we need. They can cope with multiple threads and contexts, but the current representation of contexts is limited to one dimension (the so-called *swimlanes*). UCMs display contexts in two dimensions, which is usually more understandable and which supports hierarchies of contexts.

In this subsection, we have discussed how Use Case Maps can be used to represent work. Other researchers working on approaches to help people record and represent their understanding of work include Pycock et al. (1998) and Jordan et al. (1995). More information about use-case maps can be found at http://www.usecasemaps.org.

3.4.3 Requirements Development

The process described so far that involves developing *buttons* for synchronized shadowing, performing the shadowing, detecting patterns in the data, and drawing UCMs containing work patterns, should ideally be done in an *iterative* manner as a series of studies. Each step can help improve the other steps in the subsequent iterations; for example, the work patterns can give the researchers performing synchronized shadowing a better idea of what to look for.

The final, but certainly not least important, step in our process is taking the work patterns and interpreting them so as to discover potential software requirements. The essence of this process is examining the patterns looking for signs of inefficiency such as the following:

- Frequent sequences that can be automated.
- Frequent situations where the participant jumps back and forth between contexts, and where it might be possible to allow the required activity to all occur in only one context and thus eliminate context switching.
- Situations where the participant must frequently wait, due to system delays.

- Situations where the participant frequently makes mistakes because he or she has to rely on memory to transfer information or to perform similarly mentally taxing activities.

Each of these can lead to a requirement to reduce the inefficiency through improved software.

3.5 A Case Study: Empirical Studies at Mitel

This section presents a case study in which we applied synchronized shadowing and Use Case Maps to study and model the work patterns of software maintainers at Mitel Corporation, and then develop tools to make them more productive. The Mitel software engineers we studied were working on a large telecommunications system.

Before our first synchronized shadowing sessions, we studied the software engineers enough to discover the main types of events we would provide as buttons in the synchronized shadowing tool. Table 3.1 shows the set of control keys - which we used in place of buttons - that were used by one of the two note-takers. While the first note-taker recorded the individual actions that the programmers performed, the other note-taker focused more on their high-level goals while performing their actions. The programmers were asked to think out loud while performing their task. It was the job of the second note-taker to code this information, therefore his codes focused more on hypotheses and plans.

We conducted a total of nine synchronized shadowing sessions with eight software engineers, each session lasting about an hour. We attempted to coordinate our study of each SE so that it would occur at a time when the SE was performing what he or she considered typical work with source code.

Prior to meeting the participant to begin each session, we synchronized the clocks of the two computers. We also practiced using the synchronized shadowing interface to ensure we were familiar with the coding scheme we had developed.

When the synchronized shadowing data was obtained, it was scanned manually for a short while to begin to find patterns. It was clear (as we had initially expected from earlier observations) that the vast majority of our participants' time was spent working in text editors or searching for various kinds of things.

Following the procedure outlined in Section 3.4, we worked our way through the data and developed a variety of UCMs that represent common work patterns. We found that the data coded during the initial synchronized shadowing sessions was coded at the level of detail we needed for the UCMs. Therefore we manually went back through the logs, looking at the free-form notes which had been

Table 3.1. Control sequences in Microsoft Word macros used by one of the observers during synchronized shadowing.

Control key	Description
^-v	VIS: Look at something
^-e	EDITOR: Issue an editor command
^-m	MODIFY: Write or modify some text
^-s	SEE: Issue a command in a software exploration tool
^-g	GREP: Run *grep*
^-t	TOOL: Work with some other tool.
^-u	UNIX: Type a command other than *grep*
^-z	NEW-GOAL: Start something completely new
^-space	Miscellaneous

added after each button press during synchronized shadowing. We were able to give more precise codes to each event; e.g., we needed to divide certain types of search into more detailed categories. The inter-rater reliability of this manual process was very high once we had agreed on the codes we wanted to use. For later synchronized shadowing sessions, we were able to add extra buttons to our tool and therefore reduce the need for subsequent manual analysis.The following are two examples of UCMs we generated from this work.

While exploring a series of files, our sample logs showed the participants doing three distinct types of activities: Searching for text, copying text (into the copy buffer) or merely reading the text. As for searching, it was done either using previously copied text or by manually typing the parameters. Users would jump repeatedly from file to file, using the contents of the copy buffer to transfer a piece of text from one file to use as a search parameter in another. Figure 3.7 shows the UCM we constructed to show this work pattern, in the context of a single file.

Figure 3.8 shows a second example UCM. In this case, the activity being frequently performed is searching through multiple files using *grep*. The results of the searches are then manipulated. The figure illustrates all the possible paths that occur in this activity.

Important observations we make from Figures 3.7 and 3.8, as well as other UCMs not shown here are:

- Search within files, and search across files, are fundamental operations to the maintainers.
- Transfer of information from file to file and from tool to tool was most commonly performed by copying and pasting. There were two important sources of information to place into the copy buffer: text in a file and text in a search result. There were four different destinations into which the buffer would be pasted. These are; A file that is being edited, a file name to open, or a search parameter either in an editor or for *grep*.

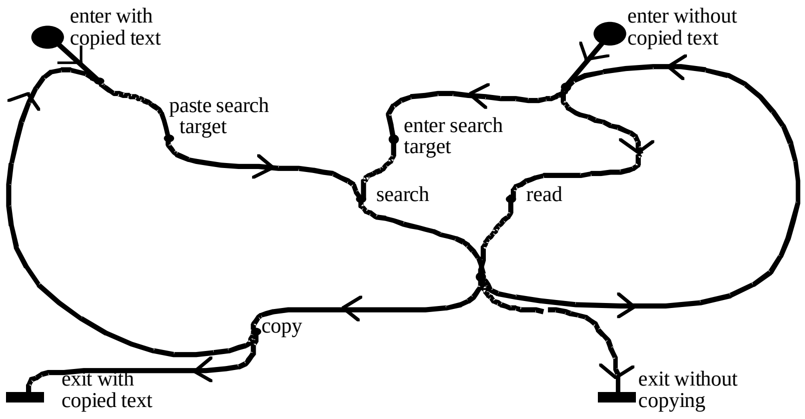


Figure 3.7. A Use Case Map showing an abstract view of the paths taken by users when exploring files (without editing).

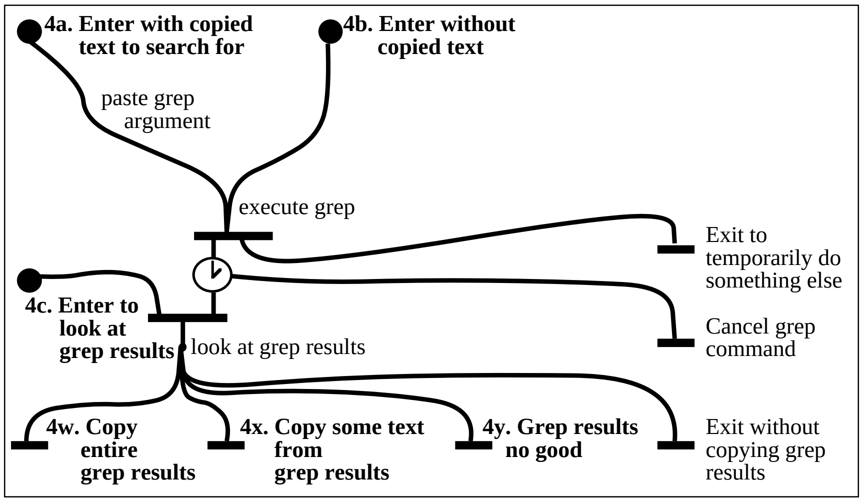

4a. Enter with copied text to search for
4b. Enter without copied text
paste grep argument
execute grep
Exit to temporarily do something else
Cancel grep command
4c. Enter to look at grep results
look at grep results
4w. Copy entire grep results
4x. Copy some text from grep results
4y. Grep results no good
Exit without copying grep results

Figure 3.8. UCM showing possible paths when performing a search using *grep* (the clock symbol represents a period of waiting).

We used a program to count the occurrences of the various categories and sequences of events in the UCMs. The results are presented in Tables 3.2 and 3.3. Of the 966 events recorded while performing synchronized shadowing, almost 30% involved searching, and almost 20% involved cutting and pasting.

Table 3.3 shows sequences where the user copies some text and then pastes it. It is clear that most of the time, when a maintainer selects some text in a file, he or she intends to use that as a search argument to find other occurrences of the text in the same file or in other files. Similarly, when a maintainer selects text in a search result, it is generally the name of a file that he or she intends to open and study in more depth.

Analysis of the data shown above leads us to derive the following requirements for a software exploration tool we are developing called TkSee (Lethbridge and Anquetil, 1997; Lethbridge and Herrera, 2000).

- The tool should have a direct way to open a file from search results (e.g., by simply selecting some result)
- **Rationale:** 71% of copy operations performed on search results were performed in order to obtain a file name to open. This requirement would eliminate a considerable number of keystrokes or mouse movement (both

Table 3.2. Frequencies of the most important categories of events.

Event types	Percent of total	Percent of subtotals	Number of events
Total number of events			966
Copy text	9.2%		89
Copy from file		66.3%	59
Copy from search results		34.8%	31
Search	28.3%		273
Search in editor (one file)		59.3%	162
Search across files		40.7%	111
Using *grep*		23.8%	65
Using other tool		16.8%	46
Study (reading)	28.5%		275
Study in editor		87.6%	241
Study search results		12.4%	34
Paste text	10.6%		102
Paste to modify text		7.8%	8
Paste to search in editor		32.4%	33
Paste to open file		22.5%	23
Paste to search across files		37.3%	38

issuing copy and commands, as well as actions required to exit the search results and enter).

- The tool should have a simple command to automatically locate occurrences of whatever is in the copy buffer.
- **Rationale:** No matter what the source of copied text, users frequently used the copy buffer as the argument when performing a search in an editor. This requirement would save many paste operations and reduce the necessity to bring up a search dialog box.
- Add a command that allows the user to search for whatever is selected in the editor without doing a copy and then a paste.
- **Rationale:** 27.1% of copy operations from a file were immediately used to search in that file. This would speed this operation.

- In dialog boxes that initiate *grep*-like searches or searches within a file, pre-fill the box that specifies the argument with whatever text has just previously been selected in the editor and/or the search results.
- **Rationale:** This would reduce the need to do some copying.

The requirements above come from the data in Tables 3.2 and 3.3; thus it might be argued that we could eliminate the Use Case Map step and go straight from patterns in synchronized shadowing data, to tables, to requirements. The Use Case Maps however, had two critical roles: Firstly, we developed them iteratively as we performed pilot studies and improved our coding scheme (the set of buttons) for synchronized shadowing. As we saw patterns emerging, we drew the UCMs to obtain an understanding of which buttons should be present in our synchronized shadowing tool. Secondly, having the UCMs allows us to better explain the requirements.

Implementing the above requirements has allowed us to improve the functionality of TkSee considerably. TkSee's overall strengths include its integration of a variety of techniques for searching through source code (including the requirements illustrated above), its ability to allow maintainers to incrementally build models of aspects of the software, and its ability to support the manipulation and saving of search results and explorations. These features include those that our Use Case Maps tell us are the activities that maintainers perform most often.

3.6 Summary and Conclusions

We have described several techniques for performing observational field studies of people at work, and analyzing the resulting data. We applied these techniques to the work of software engineers in order to develop better tools for them; how-

Table 3.3. Copy-paste transitions

	Copy from file	Copy from search results
Paste to modify text	5.1%	16.1%
Paste to search in file	27.1%	9.7%
Paste to open file	1.7%	71.0%
Paste as search argument	47.5%	3.2%
Not immediately used	18.6%	0.0%

ever, the techniques should be useful whenever one's objective is to understand work practices.

We situated our empirical study techniques in a three-dimensional space. On one axis, our techniques involve observation of people performing their everyday work; therefore, they can be called *field studies* as opposed to laboratory studies. On a second axis, our tools involve active observation involving direct contact with people as they go about their daily work, whereas other techniques might only indirectly observe people or else study the products of their work. On the third axis, our techniques involve information gathering for the purpose of constructing models; we make no attempts to run controlled experiments.

To gather data while observing software engineers at work, we use a note-taking approach as opposed to videotaping. However, since it is hard to be consistent when manually taking notes, we developed a technique we call synchronized shadowing whereby two people use clock-synchronized computers that are preprogrammed with buttons that record time-stamped annotations corresponding to different kinds of observed events. This technique allows us to gather reasonably accurate information in real-time that is already partially coded, hence analysis time is greatly reduced.

We build models from our synchronized-shadowing data using Use Case Maps (UCMs), a technique originally invented for modeling real-time systems, but which is ideally suited to model work practices. From the UCMs we can see work patterns, and from the work patterns we can deduce requirements for software tools. We call our combined approach WASS (Work Analysis from Synchronized Shadowing).

There remain some open research issues with our work: Firstly it would be nice to analyze the time consumed by the participants performing the work patterns, rather than just the sequences. We know, for example, that copying and pasting is performed very frequently, but it might be that other less time-consuming activities actually take more time. We would also like to use the technique in a wider context. Currently we have only used it in the one Mitel empirical study.

3.7 Acknowledgements

We thank the Mitel employees who participated in our studies. We also thank an anonymous reviewer for valuable suggestions and K. Teresa Khidir for helping polish the manuscript.

3.8 References

Bannon, L. (1994). Representing work in design. In L. Suchman (Ed.), *Representations of Work: A Symposium*. Monograph for proceedings of the 27th HICSS, January, Maui, Hawaii.

Bannon, L, and Bødker, S. (1991). Beyond the interface: Encountering artifacts in use. In J. Carroll (Ed.), *Designing Interaction: Psychology at the Human-Computer Interface*. Cambridge University Press: New York.

Bendifallah, S., and Scacchi, W. (1987). Understanding software maintenance work, *IEEE Transactions on Software Engineering*, 13(3), 311-323.

Blomberg, J., Suchman, L., and Trigg, R. (1996). Reflections on a work-oriented design project, *Human Computer Interaction*, 11, 237-265.

Boehm-Davis, D. Holt, R., and Schultz, A. (1992). The role of program structure in software maintenance, *Int. Journal of Man Machine Studies, 36,* 21-63.

Buhr, R.J.A (1998). Use case maps as architectural entities for complex systems, *IEEE Trans. Software Engineering*,. 24(12) Dec., 1131-1155.

Buhr, R.J.A and Casselman, R.S. (1996). *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Button, G., and Dourish, P. (1996). Technomethodology: Paradoxes and Possibilities, In *Proc CHI '96; Human Factors in Computing Systems*, Vancounver, 19-26..

Cook, T., and Campbell, D. (1979) *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Rand McNally: Chicago, IL.

Curtis, W (1986) By the way, did anyone study any real programmers? In E. Soloway and S. Iyengar (Eds.). In *Proc. Empirical Studies of Programmers*. Norwood, NJ, pp. 256-262.

Curtis, B., Krasner, H., and Iscoe, N. (1988). A field study of the software design process for large systems, *Communications of the ACM,* 31(11), 1268-1287.

D'Astous, P., and Robillard, P. (2000). Protocol analysis in software engineering studies. *Empirical Studies in Software Engineering,* Khaled El-Eman and Janice Singer, eds., MIT Press.

Denzin, N.K and Lincoln, Y.S. (1994). *Handbook of Qualitative Research*, Sage Publications: Thousand Oaks, CA.

Glaser, B., and Strauss, A., (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine deGruyter: Hawthorne, NY.

Hutchins, E. (1994). *Cognition in the Wild*, MIT Press: Cambridge, MA.

Jordan, B., Goldman, R., and Sachs, P. (1995). Tools for the workplace, *Communications of the ACM*, 38(9), 42.

Kraut, R., and Streeter, L. (1995). Coordination in software development, *Communications of the ACM*, 38(3), 69-81.

Kyng, M., and Mathiassen, L. (1997). *Computers and Design in Context.* MIT Press: Cambridge, MA.

Lethbridge, T.C. and Anquetil, N. (1997). Architecture of a source code exploration tool: A software engineering case study, *University of Ottawa, Computer Science Technical report TR-97-07.*

Lethbridge, T.C. and Herrera, F. (2000). Towards assessing the usefulness of the TkSee software exploration tool: a case study, Elsewhere in this book..

Lethbridge, T., Singer, J., Vinson, N., and Anquetil, N. (1997). An examination of software engineering work practices. In *Proc. CASCON*, Toronto, October: IBM, 209-223.

Lethbridge, T.C., Lyon, and S., Perry, P. (2000). The management of university-industry collaborations involving empirical studies of software engineering, In *Empirical Studies in Software Engineering*, Khaled El-Eman and Janice Singer, eds., MIT Press.

Litman, D., Pinto, J., Letovsky, S. and Soloway, E. (1996). Mental models and software maintenance. In *Proc. Empirical Studies of Programmers: First Workshop.*

Neuman, W., L. (1997). *Social Research Methods: Qualitative and Quantitative Aproaches*. Allyn and Bacon: Boston, MA.

Porter, A., and Votta L. (1998). Comparing detection methods for software requirements inspections: A replication using professional subjects, *Empirical Software Engineering*, 3, 355-379.

Pycock, J., Palfreyman, K., Allanson, J., and Button, G. (1998). Representing fieldwork and articulating requirements through VR, In. *Proc CSCW*, Seattle, 383-392.

Simonsen, J., and Kensing, F. (1997). Using ethnography in contextual design, *Communications of the ACM*, 40(7), 82-88.

Singer, J., and Lethbridge, T. C. (1998a) Work practices as an alternative method to assist tool design in software engineering, *International Workshop on Program Comprehension*, Ischia, Italy, 173-179.

Singer, J., and Lethbridge, T.C. (1998). Studying work practices to assist tool design in software engineering. In *6th IEEE International Workshop on Program Comprehension*, Italy, 173-179. A longer version appears as: University of Ottawa, Computer Science Technical Report TR-97-08.

Sanderson, P.M., Scott, J.J.P., Johnston, T., Mainzer, J., Watanabe, L. M. and James, J.M. (1994). MacSHAPA and the enterprise of exploratory sequential data analysis (ESDA). *International Journal of Human-Computer Studies,* 41 (5), 633-681.

Suchman, L. (1987). *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press: New York.

Suchman, L. (Ed.) (1995). Representations of work, *Special issue of the Communications of the ACM,* 38(9), 33-68.

Walz, D., Elam, J., and Curtis, B. (1993). Inside a software design team: Knowledge acquisition, sharing, and integration, *Communication of the ACM, 36(10)*, 63-77.