Studying Software Engineers: Data Collection Techniques for Software Field Studies

Article *in* Empirical Software Engineering · July 2005

DOI: 10.1007/s10664-005-1290-x · Source: DBLP

Timothy Lethbridge
University of Ottawa
200 PUBLICATIONS **4,351** CITATIONS
SEE PROFILE

Susan Sim
University of Toronto
102 PUBLICATIONS **1,761** CITATIONS
SEE PROFILE

Janice Singer
National Research Council Canada
84 PUBLICATIONS **3,751** CITATIONS
SEE PROFILE

Empirical Software Engineering, 10, 311–341, 2005.

Studying Software Engineers: Data Collection Techniques for Software Field Studies

TIMOTHY C. LETHBRIDGE tcl@site.uottawa.ca
School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario K1N 6N5, Canada

SUSAN ELLIOTT SIM ses@ics.uci.edu
Department of Informatics, University of California, Irvine, 444 Computer Science, Irvine, CA 92697-3425, USA

JANICE SINGER janice.singer@nrc-cnrc.gc.ca
National Research Council Canada, Institute for Information Technology, Montreal Rd, Building M-50, Ottawa, Ontario K1A 0R6, Canada

Editor: Marv Zelkowitz

Abstract. Software engineering is an intensely people-oriented activity, yet too little is known about how designers, maintainers, requirements analysts and all other types of software engineers perform their work. In order to improve software engineering tools and practice, it is therefore essential to conduct *field studies*, i.e., to study real practitioners as they solve real problems. To do so effectively, however, requires an understanding of the techniques most suited to each type of field study task. In this paper, we provide a taxonomy of techniques, focusing on those for data collection. The taxonomy is organized according to the degree of human intervention each requires. For each technique, we provide examples from the literature, an analysis of some of its advantages and disadvantages, and a discussion of how to use it effectively. We also briefly talk about field study design in general, and data analysis.

Keywords: Field studies, work practices, empirical software engineering.

1. Introduction

Software engineering involves real people in real environments. People create software, people maintain software, people evolve software. Accordingly, to truly understand software engineering, it is imperative to study people—software practitioners as they solve real software engineering problems in real environments. This means conducting studies in field settings.

But how does one attain this admirable goal? What techniques are available for gathering data and analyzing the results? In what situations are each of these techniques suitable? What difficulties might the software researcher encounter when performing field studies? To begin to address these questions, in this paper we make a first attempt at providing a taxonomy of data collection techniques for performing field studies. We illustrate the discussion with examples from our own work as well as numerous studies reported in the literature.

The authors appear in alphabetical order and contributed equally.

The techniques we talk about have primarily been adapted from fields such as sociology, psychology, and human–computer interaction. In particular, these methods rely heavily on the use of field study techniques. Field study techniques are a group of methods that can be used individually or in combination to understand different aspects of real world environments.

The results of such studies can be applied when one has several different types of research goals. First, they can be used to derive requirements for software tools and environments. For example, we have performed field studies where we learned about software design and maintenance, and then successfully applied our results to tool design and integration (Singer et al., 1997, 1998). Second, the results can be used to improve software engineering work practices. For example, one of us (Sim) was able to make useful recommendations following a study of how newcomers adapt to a software team (Sim and Holt, 1998). Third, analysis of the results can yield new theories or hypotheses that can then be subjected to controlled experimental validation (Seaman and Basili, 1998; Singer, 1998).

The taxonomy we have created is based on the degree of human intervention each data collection technique requires. We begin in the next section by providing an overview of the taxonomy and describing how the taxonomy can be used in selecting a technique for a field study. In Section 3, we talk about each of the techniques individually, providing examples from research where it was applied, and giving advantages and disadvantages of its use. We talk briefly in Section 4 about study design and recording and present a brief discussion of data analysis. We conclude the paper with a discussion of how to use these techniques in the most appropriate fashion.

2. Data Collection Methods

When conducting field studies it is important to obtain accurate and reliable information about the phenomenon under study. Interviews and questionnaires are the most straightforward instruments, but the data they produce typically present an incomplete picture. For example, assume your goal is to assess which programming language features are most error-prone. A developer can give you general opinions and anecdotal evidence about this; however you would obtain far more accurate information by recording and analyzing the developer's *work practices*—their efforts at repeatedly editing and compiling code. Methods such as think-aloud protocols and work diaries are used for this type of research.

To learn about different aspects of a phenomenon, it is often best to use multiple data collection methods. One then analyzes the resulting data to *triangulate* the work practice. In the remainder of this section we survey the range of techniques that can be used in this triangulation process, and discuss some of the criteria that can be used to choose techniques.

2.1. A Taxonomy

In Table 1, we present a taxonomy for the data collection techniques. Each technique is categorized according to the degree of human contact it requires. First degree contact

Table 1. Data collection techniques suitable for field studies of software engineering.

Category	Technique
	Inquisitive techniques
First Degree	• Brainstorming and Focus Groups
(direct involvement of software engineers)	• Interviews
	• Questionnaires
	• Conceptual Modeling
	Observational techniques
	• Work Diaries
	• Think-aloud Protocols
	• Shadowing and Observation Synchronized Shadowing
	• Participant Observation (Joining the Team)
Second Degree	• Instrumenting Systems
(indirect involvement of software engineers)	• Fly on the Wall (Participants Taping Their Work)
Third Degree (study of work artifacts only)	• Analysis of Electronic Databases of Work Performed
	• Analysis of Tool Use Logs
	• Documentation Analysis
	• Static and Dynamic Analysis of a System

requires direct access to a participant population. Second degree contact requires access to participants' environment as they work, but without requiring either direct access to participants, or for participants and researchers to interact. Finally, third degree contact requires access only to work artifacts, such as source code or documentation.[1]

This taxonomy is informative because the degree of contact reflects the kinds of data that can be collected, the resources required, the flexibility available to the researcher, and the reliability of the resulting information. Close contact with subjects will require a stronger working relationship than unobtrusive archival research. Given this cost, lower degree techniques should only be selected when they are the only way to produce the data needed to answer the research question, i.e., when you need to know what software engineers are thinking. These trade-offs will be explored further in the next two subsections.

In Section 3, each technique is presented, followed by a list of advantages and disadvantages, and is illustrated, where possible, using an example taken from the software engineering literature. The example is given in the context of the questions the researchers were trying to answer and how the particular technique allowed them to do so.

All the methods, from the simplest to the most complex, need to be applied with care and respect. Questions, observation sessions and other activities must be planned carefully to ensure that the data collected is meaningful. A poorly-worded question results in ambiguous responses that cannot be interpreted or analyzed. Additionally, participants must be treated with respect because they are, first and foremost, human beings in social situations. Researchers need to be sensitive to the disruptions the researcher's presence can cause (see Singer and Vinson for more on research ethics in software engineering studies).

The set of available methods is constantly evolving. A good place to look for new data gathering and analysis methods is in the human–computer interaction literature (e.g., (Kensing, 1998; Snelling and Bruce-Smith, 1997); there is much in common between trying to improve a software system so users can use it more effectively, and trying to improve software engineering practices.

2.2. Situating the Data Collection Method

Selection of a data collection method should be done in the context of a research goal or question. We propose that the research method should be chosen in a similar way to metrics in software engineering, i.e., following a method analogous to Basili's GQM (Goals Questions Metrics) approach (Basili, 1992).

The first step in designing any study is to establish a set of well-understood goals, because many of the subsequent design decisions depend on them. The goals should describe the phenomenon being studied and the purpose of the study, i.e., how the results will be used. Field studies in software engineering have had a variety of goals. Some of them try to develop tool requirements by studying software engineers (Sim et al., 1998; Singer, 1998; Singer et al., 1998). Others aim to develop a better understanding of how software engineers perform some particular activity such as maintenance (Jørgensen, 1995; Kemerer and Slaughter, 1997); many seek to go beyond mere understanding and derive recommendations for improved procedures or processes (Perry et al., 1994; Seaman and Basili, 1998; Sim and Holt, 1998; Wolf and Rosenblum, 1993).

The goals of the research drive the formulation of the research questions, which in turn drive the research design, which in turn dictate the choice of data collection technique(s). We suggest that there are three factors to consider in selecting a technique: the degree of access required to software engineers, volume of data produced, and type of research question.

Table 2 presents a summary of the data collection techniques; the second column shows the kinds of questions each can answer. The researcher can choose a set of techniques that will together answer the required questions. The third column in Table 2, indicates the amount of data. It should be noted that the degree of contact is not directly related to the amount of data collected.

These three considerations, degree of contact, type of research question, and volume of data produced, taken together suggest which type of study will be most appropriate. Case studies can be performed where the questions are broad, there is little background knowledge, and little data to comparatively analyze. Research that focuses on data analysis is suited towards quantifying a phenomenon, such as how many programmers there are and what languages they know. Many of the field studies in software engineering tend to be exploratory in nature, because we are still gathering basic knowledge about the human factors surrounding software development and maintenance. As a result, a case study design is commonly used and the study results in a theory or model that can be tested later. As our knowledge base grows, we can employ designs that test these theories or models.

Section 4 provides additional guidance on selecting and applying a data collection method. In particular, issues in record-keeping and analysis of data are considered briefly. It is important to keep in mind how the goals and questions of software engi-

Table 2. Questions asked by software engineering researchers (column 2) that can be answered by field study techniques.

Technique	Used by researchers when their goal is to understand:	Volume of data	Also used by software engineers for:
First Order Techniques			
Brainstorming and Focus Groups	Ideas and general background about the process and product, general opinions (also useful to enhance participant rapport)	Small	Requirements gathering, project planning
Surveys	General information (including opinions) about process, product, personal knowledge etc.	Small to Large	Requirements and evaluation
Conceptual modeling	Mental models of product or process	Small	Requirements
Work Diaries	Time spent or frequency of certain tasks (rough approximation, over days or weeks)	Medium	
Think-aloud sessions	Mental models, goals, rationale and patterns of activities	Medium to large	UI evaluation
Shadowing and Observation	Time spent or frequency of tasks (intermittent over relatively short periods), patterns of activities, some goals and rationale	Small	Advanced approaches to use case or task analysis
Participant observation (joining the team)	Deep understanding, goals and rationale for actions, time spent or frequency over a long period	Medium	
Second Order Techniques			
Instrumenting systems	Software usage over a long period, for many participants	Large	Software usage analysis
Fly in the wall	Time spent intermittently in one location, patterns of activities (particularly collaboration)	Medium	
Third Order Techniques			
Analysis of work databases	Long-term patterns relating to software evolution, faults etc.	Large	Metrics gathering
Analysis of tool use logs	Details of tool usage	Large	
Documentation analysis	Design and documentation practices, general understanding	Medium	Reverse engineering
Static and dynamic analysis	Design and programming practices, general understanding	Large	Program comprehension, metrics, testing, etc.

neering researchers differ from those of the software engineers themselves: software engineers' goals include improving quality of a specific product, reducing cost, and reducing time-to-market, etc. On the other hand, the goals of software engineering researchers are to understand the general principles that will help all software engineers achieve these goals. The right column in Table 2 illustrates that many of the techniques presented in this paper are in widespread use by software practitioners to help define and manage their specific project; however, researchers use the techniques to answer significantly different types of questions.

2.3. Trade-offs Between Data Collection Methods

Before describing the individual methods in the next section, some general guidelines will be provided here on choosing among them. The three categories of data collection methods vary in terms of the cost of resources required to collect the data, their reliability, their flexibility, and the phenomenon addressed. The relative merits of the different categories are shown in Figure 1.

Cost is a function of the effort required to collect the data, the record-keeping technique used, the amount of data produced, and the effort required to analyze the data. In general, lower degree techniques are more expensive to use because they require more time and effort from researchers and study participants.

Methods that produce more data require more time to analyze that data. Computer-based records are easier to analyze because software can be brought to bear on the data, more so than other data sources, such as videotapes. Some exceptions may arise when working with very large bodies of code or tool logs.

Humans tend not to be reliable reporters, as they often do not remember past events with a high degree of accuracy. Records of activities, such as tapes, work products, and repositories, tend to be more reliable. However, care must be taken when interpreting these data sources as they may not be consistent, internally or with each other.

Despite their drawbacks, first degree techniques are invaluable because of their flexibility and the phenomenon they can be used to study. Existing logs and repositories are easy to use but the data available is highly constrained. Software engineers, on the other hand, can be asked about a much wider range of topics. Second degree techniques

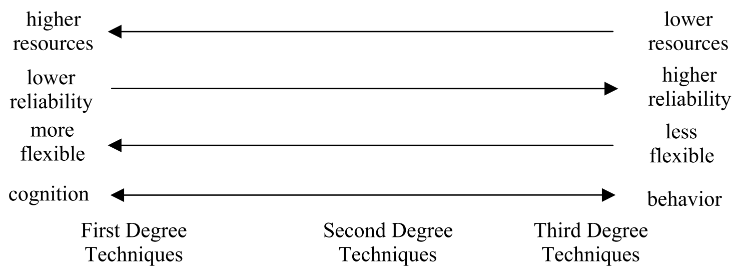

Figure 1. Cost, reliability, flexibility, and phenomena addressed.

lie somewhere in between. However, all of these techniques can still be used for exploratory questions.

Finally, some contact with software engineers is necessary to find out what they think or feel. More removed techniques can only tell you what they were doing, however, this is not a problem if actions or work practice is the main interest of the study. Some inferences can be made regarding cognition from behavior, but they need to be confirmed with direct inquiries.

In summary, each category of technique has its drawbacks, so it is necessary to use the appropriate combination to provide the data necessary to provide evidence to answer the research questions. First and second degree techniques are more complex than third degree techniques, but in many situations this is an argument in their favor.

3. Survey of Data Collection Methods

In this section, we describe the data collection techniques listed in Table 1. We use the taxonomy to organize the presentation of the techniques, beginning with first degree techniques (direct involvement of subjects), moving on to second degree techniques (indirect involvement of subjects), and concluding with third degree techniques (study of work artifacts only). Each of the techniques is described in the same way. First the technique is outlined. Then its advantages and disadvantages are identified. Finally, one or more examples of its use in software engineering research are given.

3.1. First Degree Techniques: Direct Involvement of Software Engineers

The first five techniques listed in Table 1 are what we call *inquisitive* techniques, while the remaining ones are primarily *observational*. Each type is appropriate for gathering different types of information from software engineers.

Inquisitive first-degree techniques allow the experimenter to obtain a general understanding of the software engineering process. Such techniques are probably the only way to gauge how enjoyable or motivating certain tools are to use or certain activities to perform. However, they are often subjective, and additionally do not allow for accurate time measurements.

Observational first-degree techniques provide a real-time portrayal of the studied phenomena. However, it is more difficult to analyze the data, both because it is dense and because it requires considerable knowledge to interpret correctly. Observational techniques can be used at randomly chosen times or when a software engineer is engaged in a specific type of activity (such as whenever she is using a debugger). Observational techniques always run the risk of changing the process simply by observing it; the Hawthorne effect was first identified when a group of researchers found that output was not related to environmental conditions as expected, but rather to whether or not workers were being observed (Draper, 2004; Robbins, 1994). Careful consideration of this effect is therefore warranted in implementing the research and explaining its purpose and protocol to the research participants.

3.1.1. Brainstorming and Focus Groups

In brainstorming, several people get together and focus on a particular issue. The idea is to ensure that discussion is not limited to 'good' ideas or ideas that make immediate sense, but rather to work together to uncover as many ideas as possible. Brainstorming works best with a moderator because the moderator can motivate the group and keep it focused. Brainstorming works best when there is a simple 'trigger question' to be answered and everybody is given the chance to contribute whatever comes to their mind, initially on paper. A good seminal reference for this process, called Nominal Group Technique, is the work of Delbecq et al. (1975). Trigger questions, such as, "What are the main tasks that you perform?,'' "What features would you like to see in software engineering tools?,'' or "What difficulties do you experience in your daily work?,'' can result in extensive lists of valuable ideas which can then be discussed in more detail, ranked, and analyzed.

Focus Groups are similar to brainstorming. However, focus groups occur when groups of people are brought together to focus on a particular issue (not just generate ideas). They also involve moderators to focus the group discussion and make sure that everyone has an opportunity to participate. There is a large volume of written material on how to properly design and moderate focus groups. Nielsen wrote a short article which outlines the process and some of its uses and misuses (Nielsen, 1997).

Advantages. Brainstorming and focus groups are excellent data collection techniques to use when one is new to a domain and seeking ideas for further exploration. They are good at rapidly identifying what is important to the participant population. Two important side benefits of brainstorming and focus groups are that they can introduce the researchers and participants to each other and additionally give the participants more of a sense of being involved in the research process. Conducting research in field environments is often stressful to the research participants; they are more likely to be willing participants if they feel comfortable with the researchers and feel they are partners in research that focuses on issues that they consider to be important.

Disadvantages. Unless the moderator is very well trained, brainstorming and focus groups can become too unfocused. Although the nominal group technique helps people to express their ideas, people can still be shy in a group and not say what they really think. Just because a participant population raises particular issues, this does not mean the issues are really relevant to their daily work. It is often hard to schedule a brainstorming session or focus group with the busy schedules of software engineers.

Examples. Bellotti and Bly (1996) used brainstorming during an initial meeting with a product design group. The goal of their research was to study the process of mobile collaboration among design team members. The brainstorming meeting was held to identify problems and possible solutions as seen by the team. This meeting gave the researchers an initial understanding of the team's work and additionally let the researchers know how existing technology was either supporting or inhibiting the work. A nice side effect of the meeting was that it gave the researchers an entry point for communication about the design process with their colleagues in the design department at Apple. One of us (Lethbridge) regularly uses brainstorming; we have described our technique in more detail elsewhere (Lethbridge and Laganière, 2001).

Hall and her colleagues have published a number of papers based on a large study involving focus groups to understand software process improvement (Baddoo and Hall, 2002a,b; Beecham et al., 2003; Rainer and Hall, 2003). In their study, they used 13 software companies and implemented 49 focus groups. The groups were comprised of between 4 and 6 participants. The companies were chosen based on certain characteristics, but overall were representative of the industry. Each session lasted 90 minutes. There were three types of groups: senior managers, project managers, and developers. The focus groups were moderated and tackled very specific questions aimed at understanding several factors leading to success and failure for software process improvement.

3.1.2. Interviews and Questionnaires

In this subsection, the strengths and drawbacks common to both interviews and questionnaires are discussed. Each technique is then described individually in the following two subsections.

Both interviews and questionnaires are centered on asking a series of questions. Questions can be closed-ended, i.e., multiple-choice, or they can be open-ended, i.e., conversational responses; it is best to always have at least some open-ended questions to gain information that cannot be relayed by more specific information seeking questions. To implement interviews and questionnaires effectively, questions and forms must be crafted carefully to ensure that the data collected is meaningful (DeVaus, 1996; Foddy, 1994). A poorly worded question results in ambiguous responses that cannot be interpreted or analyzed. It is highly advisable to pilot test the questions or forms and then re-design them as you learn which questions unambiguously attack the pertinent issues.

In order to generate good statistical results from interviews or a questionnaire, a sample must be chosen that is representative of the population of interest. This requirement is particularly difficult in software engineering because we lack good demographic information about the population of developers and maintainers. However, this drawback should not prevent us from using interviews and questionnaires to conduct field studies, if we do not intend to perform statistical tests on the data or when the problem or population is small and well-defined.

Interviews and questionnaires are often conducted in the same series of studies, with the interviews providing additional information to the answers from the questionnaires.

Advantages. People are familiar with answering questions, either verbally or on paper, and as a result they tend to be comfortable and familiar with this data collection method. Participants also enjoy the opportunity to answer questions about their work.

Disadvantages. Interviews and questionnaires rely on respondents' self-reports of their behaviors or attitudes. This dependency can bias the results in a number of ways. People are not perfect recorders of events around them; in particular, they preferentially remember events that are meaningful to them. For instance in one of our questionnaires, participants reported that reading documentation was a time-consuming aspect of their job, but in 40 hours of observation, we hardly saw anyone doing so.[2]

If the objective of interviews and questionnaires is to obtain statistics based on the answers to fixed questions, then issues of sampling arise. Most studies in software engineering have to use what is called convenience sampling, meaning that we involve whoever is available and volunteers. This will result in various types of bias, such as self-selection bias (those most interested in our work may have different characteristics from the population as a whole). Results must always therefore be reported with an acknowledgement of potential biases, and other threats to validity. And results should be used keeping the biases in mind. In most cases, slightly biased data is still much more useful than a complete lack of data.

3.1.3. Interviews

Face-to-face interviews involve at least one researcher talking, in person, to at least one respondent at a time. Normally, a fixed list of carefully worded questions forms the basis of the interview. Depending on the goal of the study, respondents may be encouraged to elaborate on areas and deviate slightly from the script.

Telephone interviews are the middle ground between face-to-face interviews and questionnaires. You have the interactivity of an interview at the cost and convenience of a phone call. Telephone interviews are not as personal as face-to-face interviews, yet they still provide researchers with opportunities to clarify questions and further probe interesting responses. Although this technique is popular in opinion polling and market research, it is little used in empirical software engineering.

Advantages. Interviews are highly interactive. Researchers can clarify questions for respondents and probe unexpected responses. Interviewers can also build rapport with a respondent to improve the quality of responses.

Disadvantages. Interviews are time and cost inefficient. Contact with the respondent needs to be scheduled and at least one person, usually the researcher, needs to travel to the meeting (unless it is conducted by phone—but this lessens the rapport that can be achieved). If the data from interviews consists of audio or video tapes, this needs to be transcribed and/or coded; careful note-taking may, however, often be an adequate substitute for audio or video recording.

Examples. Interviews have been used in many studies because they fit well with many types of inquiries. We have used interviews in longitudinal studies as an aid in understanding how newcomers adapt to a development team and software system (Sim and Holt, 1998). We interviewed newcomers once every three weeks over a number of months to track their progress as maintenance team members. Since this was an exploratory study, the questions were open-ended.

Curtis et al. (1988) used interviews to study the design process used on 19 different projects at various organizations. They interviewed personnel from three different levels of the participating projects, systems engineers, senior software designers and project managers. The researchers conducted 97 interviews, which resulted in over 3000 pages of transcripts of the audio recordings. They found three key problems common to the design processes: communication and coordination breakdowns, fluctuating and conflicting product requirements, and the tendency for application domain knowledge to be located in individuals across the company. They characterized the problems at each level of a model they subsequently defined.

Damian et al. (2004) used interviews of experienced personnel and senior management to examine how changes in the requirements engineering process affected software development practice. Because there was limited historical data on the previous requirements process, the interviews were key to provide information on how the changes were affecting the current practice. In addition to the initial interviews, follow-up interviews were conducted after a questionnaire to elucidate the responses. Overall, Damian et al. found the improved requirements process was useful to the product development team in that it resulted in better documentation of requirements, better understanding of the market need, and better understood requirements. However, better communication, collaboration and involvement of other stakeholder groups is still required.

3.1.4. Questionnaires

Questionnaires are sets of questions administered in a written format. These are the most common field method because they can be administered quickly and easily. However, very careful attention needs to be paid to the wording of the questions, the layout of the forms, and the ordering of the questions in order to ensure valid results. Pfleeger and Kitchenham have published a six-part series on principles of survey research starting with (Pfleeger and Kitchenham, 2001). This series gives detailed information about how to design and implement questionnaires. Punter et al. (2003) further provides information on conducting web-based surveys in software engineering research.

Advantages. Questionnaires are time and cost effective. Researchers do not need to schedule sessions with the software engineers to administer them. They can be filled out when a software engineer has time between tasks, for example, waiting for information or during compilation. Paper form-based questionnaires can be transported to the respondent for little more than the cost of postage. Web-based questionnaires cost even less since the paper forms are eliminated and the data are received in electronic form. Questionnaires can also easily collect data from a large number of respondents in geographically diverse locations.

Disadvantages. Since there is no interviewer, ambiguous and poorly-worded questions are problematic. Even though it is relatively easy for software engineers to fill out questionnaires, they still must do so on their own and may not find the time. Thus, return rates can be relatively low which adversely affects the representativeness of the sample. We have found a consistent response rate of 5% to software engineering surveys, when people are contacted personally by email and asked to complete a web-based survey. If the objective of the questionnaire is to gather data for rigorous statistical analysis in order to refute a null hypothesis, then response rates much higher than this will be needed. However, if the objective is to understand trends, with reasonable confidence, then low response rates may well be fine. The homogeneity of the population, and the sampling technique used also affect the extent to which one can generalize the results of surveys. In addition to the above, responses tend to be more terse than with interviews.

Examples. One of us (Lethbridge) used questionnaires (Lethbridge, 2000) that were partly web-based and partly paper-based to learn what knowledge software engineers apply in their daily work, and how this compares to what they were taught in their formal education. Respondents were asked four questions about each of a long list of topics. We ran several pilot studies for the questionnaires, but nevertheless found upon analyzing the data that a couple of the topics[3] were interpreted in different ways by different respondents. Despite this, we were able to draw many useful conclusions about how software engineers should be trained.

Iivari used a paper-based questionnaire to test nine hypotheses about factors affecting CASE tool adoption in 52 organizations in Finland (Iivari, 1996). The author contacted organizations who had purchased CASE tools and surveyed key information systems personnel about the use of the tool. Companies and individuals were more likely to use CASE tools when adoption was voluntary, the tool was perceived to be superior to its predecessor(s) and there was management support.

In the example cited above, Damian et al. (2004) also used questionnaires to obtain information from the product team. All sources of information from the study were combined to triangulate the data and thus contribute to greater internal validity.

One of us (Sim) used a web-based questionnaire to study source code searching behaviors (Sim et al., 1998). We solicited respondents from seven Usenet newsgroups from the `comp.*` hierarchy to complete a questionnaire at a given web address. The questionnaire used a two-page format. The first page informed the participants of the goals of the study and their rights and the second page displayed the questions. Using a combination of open- and closed-ended questions, we identified a set of eleven search archetypes.

3.1.5. Conceptual Modeling

During conceptual modeling, participants create a model of some aspect of their work—the intent is to bring to light their mental models. In its simplest form, participants draw a diagram of some aspect of their work. For instance, software engineers may be asked to draw a data flow diagram, a control flow diagram or a package diagram showing the important architectural clusters of their system. As an orthogonal usage, software engineers may be asked to draw a physical map of their environment, pointing out who they talk to and how often.

A more sophisticated version of conceptual modeling involves the use of a tool such as a CASE tool, or a tool specially designed for conceptual modeling. The researchers may ask the participants to create models from scratch, or they may create the diagrams and then ask the participants to either confirm them or else suggest modifications.

Advantages. System illustrations provide an accurate portrayal of the user's conception of his or her mental model of the system. Such models are easy to collect and require only low-tech aids (pen and paper).

Disadvantages. The results of conceptual modeling are hard to interpret, especially if the researcher does not have domain knowledge about the system. Some software engineers are reluctant to draw, and the quality and level of details in diagrams can vary significantly.

Examples. Scacchi (2003) describes an initial effort to formally model development processes in an open source development project. Scacchi describes a rich picture model of one type of open source project that links the various roles, artifacts and tools to the development process overall. Additionally, Scacchi describes a formal model that was built using a process modeling language. The model was built using a process meta-model as its semantic foundation. The goal of the modeling was to provide a formal computational understanding that could be analysed, shared, and compared within the research and practice communities.

In one of our studies, we collected system maps from all members of the researched group. Additionally, as we followed two newcomers to a system, we had them update their original system maps on a weekly basis. We gave them a photocopy of the previous week's map, and asked them to either update it or draw a new one. The newcomers almost exclusively updated the last week's map.

In our group study, our instructions to the study participants were to 'draw their understanding of the system.' These instructions turned out to be too vague. Some participants drew data flow diagrams, some drew architectural clusters, others listed the important data structures and variables, etc. Not surprisingly, the manager of the group subsequently noted that the system illustrations reflected the current problems on which the various software engineers were working.

We learned from this exercise that for illustration data to be useful, it is important to specify to the greatest extent possible the type of diagram required. It is next to impossible to compare diagrams from different members of a group if they are not drawing the same type of diagram. Of course, this limits researchers in the sense that they will not be getting unbiased representations of a system. Specifying that data-flow diagrams are required means that software engineers must then think of their system in terms of data-flow.

In another project (Sayyad-Shirabad et al., 1997), we wanted to discover the concepts and terminology that several software engineers use to describe a software system. We extracted a set of candidate technical terms (anything that was not a common English word) from source code comments and documentation. Then we designed a simple program that would allow software engineers to manipulate the concepts, putting them into groups and organizing them into hierarchies. We presented the combined results to the software engineers and then iteratively worked with them to refine a conceptual hierarchy. Although there were hundreds of concepts in the complex system, we learned that the amount of work required to organize the concepts in this manner was not large.

3.1.6. Work Diaries

Work diaries require respondents to record various events that occur during the day. It may involve filling out a form at the end of the day, recording specific activities as they occur, or noting whatever the current task is at a pre-selected time. These diaries may be kept on paper or in a computer. Paper forms are adequate for recording information at the end of the day. A computer application can be used to prompt users for input at random times.

Advantages. Work diaries can provide better self-reports of events because they record activities on an ongoing basis rather than in retrospect. Random sampling of events gives researchers a way of understanding how software engineers spend their day without undertaking a great deal of observation or shadowing.

Disadvantages. Work diaries still rely on self-reports; in particular, those that require participants to recall events can have significant problems with accuracy. Another problem with work diaries is that they can interfere with respondents as they work. For instance, if software engineers have to record each time they go and consult a colleague, they may consult less often. They may also forget or neglect to record some events and may not record at the expected level of detail.

Examples. Wu et al. (2003) were interested in collaboration at a large software company. In addition to observations and interviews, they asked software engineers to record their communication patterns for a period of one day. The researchers were interested in both the interaction between the team members, and the typical communication patterns of developers. They found that developers communicate frequently and extensively, and use many different types of communication modalities, switching between them as appropriate. They also found that communication patterns vary widely amongst individual developers.

As another example, Jørgensen randomly selected software maintainers and asked them to complete a form to describe their next task (Jørgensen, 1995). These reports were used to profile the frequency distribution of maintenance tasks. Thirty-three hypotheses were tested and a number of them were supported. For example, programmer productivity (lines of code per unit time) is predicted by the size of the task, type of the change, but it is not predicted by maintainer experience, application age, nor application size.

As a slight modification of the work diary, Shull et al. (2000) asked students to submit weekly progress reports on their work. The progress reports included an estimate of the number of hours spent on the project, and a list of functional requirements begun and completed. Because the progress reports had no effect on the students' grades, however, Shull et al. found that many teams opted to submit them only sporadically or not at all.

3.1.7. Think-Aloud Protocols

In think-aloud protocol analysis (Ericcson and Simon, 1984), researchers ask participants to think out loud while performing a task. The task can occur naturally at work or be predetermined by the researcher. As software engineers sometimes forget to verbalize, experimenters may occasionally remind them to continue thinking out loud. Other than these occasional reminders, researchers do not interfere in the problem solving process. Think-aloud sessions generally last no more than two hours.

Think-aloud protocol analysis is most often used for determining or validating a cognitive model as software engineers do some programming task. For a good review of this literature, see von Mayrhauser and Vans (1995).

Advantages. Asking people to think aloud is relatively easy to implement. Additionally, it is possible to implement think-aloud protocol analysis with manual record keeping (Miles, 1979) obliterating the need for transcription. This technique gives a unique view of the problem solving process and additionally gives access to mental model. It is an established technique.

Disadvantages. Think aloud protocol analysis was developed for use in situations where a researcher could map out the entire problem space. It’s not clear how this method translates to other domains where it is impossible to know a priori what the problem space is. However, Chi (1997) has defined a technique called Verbal Analysis that does address this problem. In either case, though, even using manual record keeping, it is difficult and time-consuming to analyze think-aloud data.

Examples. Von Mayrhauser and Vans (1993) asked software developers to think aloud as they performed a maintenance task which necessitated program comprehension. Both software engineers involved in the experiment chose debugging sessions. The think-aloud protocols were coded to determine if participants were adhering to the ‘Integrated meta-model’ of program comprehension these researchers have defined. They found evidence for usage of this model, and were therefore able to use the model to suggest tool requirements for software maintenance environments.

As another example of think-aloud protocol analysis, Seaman et al. (2003) were interested in evaluating a user interface for a prototype management system. They asked several subjects to choose from a set of designated problems and then solve the problem using the system. The subjects were asked to verbalize their thoughts and motivations while working through the problems. The researchers were able to identify positive and negative aspects of the user interface and use this information in their evolution of the system.

Hungerford et al. (2004) adopted an information processing framework in using protocol analysis to understand the use of software diagrams. The framework assumes that the human cognitive processes are represented by the contents of short term memory which is then available to be verbalized during a task. The verbal protocols were coded using a pre-established coding scheme. Intercoder reliability scores were used to ensure consistency of codings across raters and internal validity of the coding scheme. Hungerford et al. found individual differences in search strategies and defect detection rates across developers. They used their findings to suggest possible training and detection strategies for developers looking for defects.

3.1.8. Shadowing/Observation

In shadowing, the experimenter follows the participant around and records their activities. Shadowing can occur for an unlimited time period, as long as there is a willing participant. Closely related to shadowing, observation occurs when the experimenter observes software engineers engaged in their work, or specific experiment-related tasks, such as meetings or programming. The difference between shadowing and observation is that the researcher shadows one software engineer at a time, but can observe many at one time.

Advantages. Shadowing and observation are easy to implement, give fast results, and require no special equipment.

Disadvantages. For shadowing, it is often hard to see what a software engineer is doing, especially when they are using keyboard shortcuts to issue commands and working quickly. However, for the general picture, e.g., knowing they are now debugging, shadowing does work well. Observers need to have a fairly good understanding of the environment to interpret the software engineer's behavior. This can sometimes be offset by predefining a set of categories or looked-for behaviors. Of course, again, this limits the type of data that will be collected.

Examples. We have implemented shadowing in our work in two ways (Singer et al., 1997). First, one experimenter took paper-and-pencil notes to indicate what the participant was doing and for approximately how long. This information gave us a good general picture of the work habits of the software engineers. We also developed a method we call *synchronized shadowing*. Here we used two experimenters and, instead of pencil and paper, used two laptop computers to record the software engineer's actions. One of us was responsible for ascertaining the participants' high level goals, while the other was responsible for recording their low-level actions. We used pre-defined categories (Microsoft Word macros) to make recording easier. Wu et al. (2003) also used pre-defined categories to shadow software engineers. However, they used a PDA based database that was easy to use and record actions.

Perry et al. (1994) also shadowed software engineers as they went about their work. They recorded continuous real-time non-verbal behavior in small spiral notebooks. Additionally, at timed intervals they asked the software engineers "What are you doing now?" At the end of each day, they converted the notebook observations to computer files. The direct observations contributed to Perry et al.'s understanding of the software process. In particular, shadowing was good for observing informal communication in the group setting.

As an example of observation, Teasley et al. (2002), were interested in whether co-locating team members affects development of software. In addition to interviews and questionnaires, they observed teams, conference calls, problem solving, and photographed various artifacts. The researchers found that satisfaction and productivity increased for co-located teams.

3.1.9. Participant Observation (Joining the Team)

In the Participant-Observer method, the researcher essentially becomes part of the team and participates in key activities. Participating in the software development process provides the researcher with a high level of familiarity with the team members and the tasks they perform. As a result, software engineers are comfortable with the researcher's presence and tend not to notice being observed.

Advantages. Respondents are more likely to be comfortable with a team member and to act naturally during observation. Researchers also develop a deeper understanding of software engineering tasks after performing them in the context of a software engineering group.

Disadvantages. Joining a team is very time consuming. It takes a significant amount of time to establish true team membership. Also, a researcher who becomes too involved may lose perspective on the phenomenon being observed.

Examples. Participant-Observer was one of the methods used by Seaman and Basili in their studies of how communication and organization factors affect the quality of software inspections (Seaman and Basili, 1998). One of the authors (Seaman) was integrated into a newly formed development team. Over seventeen months, Seaman participated in twenty-three inspection meetings. From her participation, Seaman and Basili developed a series of hypotheses on how factors such as familiarity, organizational distance, and physical distance are related to how much time is spent on discussion and tasks.

Porter et al. also used the participant-observer method (Porter et al., 1997). One of the researchers, a doctoral student, joined the development team under study as a means of tracking an experiment's progress, capturing and validating data, and observing inspections. Here, the field study technique was used in the service of more traditional experimental methods.

3.2. Second Degree Techniques: Indirect Involvement of Software Engineers

Second degree techniques require the researcher to have access to the software engineer's environment. However, the techniques do not require *direct* contact between the participant and researcher. First the data collection is initiated, then the software engineers go about their normal work, and finally the researchers return to collect the data that has been automatically gathered. As a result, these techniques require very little or no time from the software engineers and are appropriate for longitudinal studies.

3.2.1. Instrumenting Systems

This technique requires "instrumentation" to be built into the software tools used by the software engineer. This instrumentation is used to record information automatically about the usage of the tools. Instrumentation can be used to monitor how frequently a tool or feature is used, patterns of access to files and directories, and even the timing underlying different activities. This technique is also called system monitoring.

In some cases, instrumentation merely records the commands issued by users. More advanced forms of instrumentation record both the input and output in great detail so that the researcher can effectively play back the session. Others have proposed building a new set of tools with embedded instruments to further constrain the work environment (Buckley and Cahill, 1997).

Advantages. System monitoring requires no time commitment from software engineers. Since, people tend to be very poor judges of factors such as relative frequency and duration of the various activities they perform, this method can be used to provide such information accurately.

Disadvantages. It is difficult to analyze data from instrumented systems meaningfully; that is, it is difficult to determine software engineers' thoughts and goals from a series of tool invocations. The instrumentation might tell one what a software engineer was doing, but they do not expose the thinking behind that action. This problem is particularly relevant when the working environment is not well understood or constrained. For example, software engineers often customize their environments by adding scripts and macros (e.g., in `emacs`). One way of dealing with this disadvantage is to play back the events to a software engineer and ask them to comment. Although in many jurisdictions, employers have the right to monitor employees, there are ethical concerns if researchers become involved in monitoring software engineers without their knowledge.

Examples. Budgen and Thomson (2003) used a logging element when assessing how useful a particular CASE tool was. The logger element recorded data whenever an event occurred. Events were predetermined before. Textual data was not recorded. The researchers found that recording events only was a shortcoming of their design. It would have been more appropriate to collect information about the context of the particular event.

As another example, Walenstein (2003) used VNC (Virtual Network Computing) to collect verbatim screen protocols (continuous screen captures) of software developers engaged in software development activities. Walenstein also collected verbal protocols and used a theory-based approach to analyse the data.

3.2.2. Fly on the Wall (Participants Recording Their Own Work)

"Fly on the Wall'' is a hybrid technique. It allows the researcher to be an observer of an activity without being present. Participants are asked to video- or audiotape themselves when they are engaged in some predefined activity.

Advantages. The fly-on-the-wall method requires very little time from the participants and is very unobtrusive. Although there may be some discomfort in the beginning, it fades quickly.

Disadvantages. The participants may forget to turn on the recording equipment at the appropriate time and as a result the record may be incomplete or missing. The camera is fixed, so the context of what is recorded may be hard to understand. There is a high cost to analyzing the resulting data.

Examples. Berlin asked mentors and apprentices at a software organization to audiotape their meetings in order to study how expertise is passed on (Berlin, 1993). She later analyzed these recordings for patterns in conversations. She found that discussions were highly interactive in nature, using techniques such as confirmation and re-statement to verify messages. Mentors not only explain features of the system; they also provide design rationale. While mentoring is effective, it is also time-consuming, so Berlin makes some suggestions for documentation and proposes short courses for apprentices.

Walz et al. had software engineers videotape team meetings during the design phase of a development project (Walz et al., 1993). Researchers did not participate in the meetings and these tapes served as the primary data for the study. The goal of the study was to understand how teamwork, goals, and design evolved over a period of four months. Initially the team focused on gathering knowledge about the application domain, then on the requirements for the application, and finally on design approaches. The researchers also found that the team failed to record much of the key information; as a result they re-visited issues that had been settled at earlier meetings.

Robillard et al. (1998) studied interaction patterns among software engineers in technical review meetings. The software engineers merely had to turn on a videotape recorder whenever they were conducting a meeting. The researchers analyzed transcripts of the sessions and modeled the types of interactions that took place during the meetings. Their analysis led to recommendations for ways in which such meetings can be improved.

3.3. Third Degree Techniques: Analysis of Work Artifacts

Third degree techniques attempt to uncover information about how software engineers work by looking at their output and by-products. Examples of their output are source code, documentation, and reports. By-products are created in the process of doing work, for example work requests, change logs and output from configuration management and build tools. These repositories, or archives, can serve as the primary information source. Sometimes researchers recruit software engineers to assist in the interpretation or validation of the data. There are some advantages and disadvantages that are common to all third degree techniques.

Advantage. Third degree techniques require almost no time commitment from software engineers.

Disadvantages. The data collected is somewhat removed from the actual development process; older data may relate to systems or processes that have since been significantly changed, and recent data only captures a very narrow view of software development. Also, it may be difficult to interpret the data meaningfully—documentation can be hard to understand, and logs tend to contain cryptic abbreviations and comments that were only expected to form a consistent picture in the minds of the software engineers who originally wrote and read them. Due to the above, third degree techniques must normally be supplemented by other techniques to achieve research goals.

3.3.1. Analysis of Electronic Databases of Work Performed

In most large software engineering organizations, the work performed by developers is carefully managed using problem reporting, change request and configuration management systems. The copious records normally left by such systems are a rich source of information for software engineering researchers. Some of the information is recorded automatically, e.g., whenever modules are checked in and out of a library; the remainder of the information is descriptions and comments entered manually by the software engineers.

Advantages. A large amount of data is often readily available. The data is stable and is not influenced by the presence of researchers.

Disadvantages. There may be little control over the quantity and quality of information manually entered about the work performed. For example, we found that descriptive fields are often not filled in, or are filled in different ways by different developers. It is also difficult to gather additional information about a record, especially if it is very old or the software engineer who worked on it is no longer available.

Examples. Work records can be used in a number of ways. Pfleeger and Hatton analyzed reports of faults in an air traffic control system to evaluate the effect of adding formal methods to the development process (Pfleeger and Hatton, 1997). Each module in the software system was designed using one of three formal methods or an informal method. Although the code designed using formal methods tended to have fewer faults, the results were not compelling even when combined with other data from a code audit and unit testing.

Kemerer and Slaughter wanted to test a model of relationships between types of repairs performed during maintenance and attributes of the modules being changed

(Kemerer and Slaughter, 1997). Their analysis was based on the change history that contained information about creation date and author, the function of the module, the software engineer making the change, and a description of the change. They found support for their regression models and posit that software maintenance activity follows predictable patterns. For example, modules that are more complex, relatively large and old need to be repaired more often, whereas, modules with important functionality tend to be enhanced more often.

Researchers at NASA (1998) studied data from various projects in their studies of now to effectively use COTS (commercial off-the-shelf software) in software engineering. They developed an extensive report recommending how to improve processes that use COTS.

In our research, we are analyzing change logs to build models of which sections of code tend to be changed together: When a future maintainer changes one piece of code, this information can be used to suggest other code that perhaps ought also be looked at (Sayyad Shirabad et al., 2003).

Mockus et al. (2002) used data from email archives (amongst a number of different data sources) to understand processes in open source development. Because the developers rarely, if ever, meet face-to-face, the developer email list contains a rich record of the software development process. Mockus, et al. wrote Perl scripts to extract information from the email archives. This information was very valuable in helping to clarify how development in open source differs from traditional methods.

A researcher can use change requests to help guide first-degree research. For example, a researcher can review a recently completed work request with the software engineer to document problems encountered or strategies used along the way.

3.3.2. Analysis of Tool Logs

Many software systems used by software engineers generate logs of some form or another. For example, automatic building tools often leave records, as do license servers that control the use of CASE tools. Some organizations build sophisticated logging into a wide spectrum of tools so they can better understand the support needs of the software engineers.

Such tool logs can be analyzed in the same way tools that have been deliberately instrumented by the researchers—the distinction is merely that for this third-degree technique, the researchers don't have control over the kind of information collected. This technique is also similar to analysis of databases of work performed, except that the latter includes data manually entered by software engineers.

The analysis of tool logs has become a very popular area of research within software engineering. Besides the examples provided below, see Hassan, et al.'s (2004) report on the International Workshop on Mining Software Repositories.

Advantages. The data is already in electronic form, making it easier to code and analyze. The behaviour being logged is part of software engineers normal work routine.

Disadvantage. Companies tend to use different tools in different ways, so it is difficult to gather data consistently when using this technique with multiple organizations.

Examples. Wolf and Rosenblum (1993) analyzed the log files generated by build tools. They developed tools to automatically extract information from relevant events from these files. This data was input into a relational database along with the information gathered from other sources.

In one of our studies (Singer et al., 1997) we looked at logs of tool usage collected by a tools group to determine which tools software engineers throughout the company (as opposed to just the group we were studying) were using the most. We found that search and Unix tools were used particularly often.

Herbsleb and Mockus (2003) used data generated by a change management system to better understand how communication occurs in globally distributed software development. They used several modeling techniques to understand the relationship between the modification request interval and other variables including the number of people involved, the size of the change, and the distributed nature of the groups working on the change. Herbsleb and Mockus also used survey data to elucidate and confirm the findings from the analysis of the tool logs. In general they found that distributed work introduces delay. They propose some mechanisms that they believe influence this delay, primarily that distributed work involves more people, making the change requests longer to complete.

3.3.3. Documentation Analysis

This technique focuses on the documentation generated by software engineers, including comments in the program code, as well as separate documents describing a software system. Data collected from these sources can also be used in re-engineering efforts, such as subsystem identification. Other sources of documentation that can be analyzed include local newsgroups, group e-mail lists, memos, and documents that define the development process.

Advantages. Documents written about the system often contain conceptual information and present a glimpse of at least one person's understanding of the software system. They can also serve as an introduction to the software and the team. Comments in the program code tend to provide low-level information on algorithms and data. Using the source code as the source of data allows for an up-to-date portrayal of the software system.

Disadvantages. Studying the documentation can be time consuming and it requires some knowledge of the source. Written material and source comments may be inaccurate.

Examples. The ACM SIGDOC conferences contain many studies of documentation. In the conceptual modeling project mentioned earlier (Sayyad-Shirabad et al., 1997), we created our initial list of concepts in part by processing documentation. The result was used to build a code exploration system in which the technical terms became hyperlinks. This enabled people to find similar code and relevant documentation.

3.3.4. Static and Dynamic Analysis of a System

In this technique, one analyzes the code (static analysis) or traces generated by running the code (dynamic analysis) to learn about the design, and indirectly about how software engineers think and work. One might compare the programming or architectural styles of several software engineers by analyzing their use of various constructs, or the values of various complexity metrics.

Advantages. The source code is usually readily available and contains a very large amount of information ready to be mined.

Disadvantages. To extract useful information from source code requires parsers and other analysis tools; we have found such technology is not always mature—although parsers used in compilers are of high quality, the parsers needed for certain kinds of analysis can be quite different, for example they typically need to analyze the code *without* it being pre-processed. We have developed some techniques for dealing with this surprisingly difficult task (Somé and Lethbridge, 1998). Analyzing old legacy systems created by multiple programmers over many years can make it hard to tease apart the various independent variables (programmers, activities etc.) that give rise to different styles, metrics etc.

Examples. Keller et al. (1999) use static analysis techniques involving template-matching to uncover design patterns in source code—they point out, "... that it is these patterns of thought that are at the root of many of the key elements of large-scale software systems, and that, in order to comprehend these systems, we need to recover and understand the patterns on which they were built.''

Williams et al. (2000) were interested in the value added by pair programming over individual programming. As one of the measures in their experiment, they looked at the number of test cases passed by pairs versus individual programmers. They found that the pairs generated higher quality code as evidence by a significantly higher number of test cases passed.

We are using both static and dynamic analysis to discover various ways in which a system can be more easily understood. We analyze the source code to create clusters which can help the maintainer visualize a poorly structured system (Anquetil and Lethbridge, 1999; Lethbridge and Anquetil, 2000).

4. Applying the Methods

In the previous section, we described a number of diverse techniques for gathering information in a field study. The utility of data collection techniques becomes apparent when they can help us to understand a particular phenomenon. In this section, we explain how these methods can be effectively applied in an empirical study of software engineering. Some of the issues we deal with are: how to choose a data collection method, how to record the data, and how to analyze the data.

4.1. Record-Keeping Options

First degree contact generally involves one of the following three data capture methods: videotape, audiotape, or manual record keeping. These methods can be categorized as belonging to several related continua. First, they can be distinguished with respect to the completeness of the data record captured. Videotape captures the most complete record, while manual record keeping captures the least complete record. Second, they can be categorized according to the degree of interference they invoke in the work environment. Videotaping invokes the greatest amount of interference, while manual recording keeping invokes the least amount of interference. Finally, these methods can be distinguished with respect to the time involved in using the captured data. Again, videotape is the most time-intensive data to use and interpret, while manual record keeping is the least time-intensive data to use and interpret.

The advantage of videotape is that it captures details that would otherwise be lost, such as gestures, gaze direction, etc.[4] However, with respect to video recording, it is important to consider the video camera's frame of reference. Videotape can record only where a video camera is aimed. For instance, consider videotaping a software engineer to follow his eye movements. To accomplish this, it is necessary to have coordinated videotaping: one camera capturing the software engineer's back and computer screen[5]; the other camera capturing his eye movements as he or she looks at the screen. Moving the video camera a bit to the right or a bit to the left may cause a difference in the recorded output and subsequently in the interpretation of the data. Another difficulty with videotape is that video formats are generally of far poorer resolution than that of computer screens—thus it is hard to capture enough of what happens on the screen.

Audiotape allows for a fairly complete record in the case of interviews, however details of the physical environment and interaction with it will be lost. Audiotape does allow, however, for the capture of tone. If a participant is excited while talking about a new tool, this will be captured on the audio record.

Manual record keeping is the most data sparse method and hence captures the least complete data record, however manual record keeping is also the quickest, easiest, and least expensive method to implement. Manual record keeping works best when a well-trained researcher identifies certain behaviors, thoughts, or concepts during the data collection process. Related to manual record keeping, Wu et al. (2003) developed a data collection technique utilizing a PDA. On the PDA they had predetermined categories of responses that were coded each time a particular behaviour was observed. The data were easily transported to a database on a PC for further analysis.

All three data capture methods have advantages or disadvantages. The decision of which to use depends on many variables, including privacy at work, the participant's degree of comfort with any of the three measures, the amount of time available for data collection and interpretation, the type of question asked and how well it can be formalized, etc. It is important to note that data capture methods will affect the information gained and the information that it is possible to gain. But again, these methods are not mutually exclusive. They can be used in conjunction with each other.

4.2. Coding and Analyzing the Data

Field study techniques produce enormous amounts of data—a problem referred to as an "attractive nuisance'' (Miles, 1979). The purpose of this data is to provide insight into the phenomenon being studied. To meet this goal, the body of data must be reduced to a comprehensible format. Traditionally, this is done through a process of coding. That is, using the goals of the research as a guide, a scheme is developed to categorize the data. These schemes can be quite high level. For instance, a researcher may be interested in noting all goals stated by a software engineer during debugging. On the other hand the schemes can be quite specific. A researcher may be interested in noting how many times `grep` was executed in a half-hour programming session.

Audio and videotape records are usually transcribed before categorization, although transcription is often not necessary. Transcription requires significant cost and effort, and may not be justified for small, informal studies. Having made the decision to transcribe, obtaining an accurate transcription is challenging. A trained transcriber can take up to 6 hours to transcribe a single hour of tape (even longer when gestures, etc. must be incorporated into the transcription). An untrained transcriber (especially in technical domains) can do such a poor job that it takes researchers just as long to correct the transcript. While transcribing has its problems, online coding of audio or videotape can also be quite time inefficient as it can take several passes to produce an accurate categorization. Additionally, if a question surfaces later, it will be necessary to listen to the tapes again, requiring more time.

Once the data has been categorized, it can be subjected to a quantitative or qualitative analysis. Quantitative analyzes can be used to provide summary information about the data, such as, on average, how often `grep` is used in debugging sessions. Quantitative analyzes can also determine whether particular hypotheses are supported by the data, such as whether high-level goals are stated more frequently in development than in maintenance.

When choosing a statistical analysis method, it is important to know whether your data is consistent with assumptions made by the method. Traditional, inferential statistical analyzes are only applicable in well-constrained situations. The type of data collected in field studies often requires *nonparametric* statistics. Nonparametric statistics are often called "distribution-free'' in that they do not have the same requirements regarding the modeled distribution as parametric statistics. Additionally, there are many nonparametric tests based on simple rankings, as opposed to strict numerical values. Finally, many nonparametric tests can be used with small samples. For more information about nonparametric statistics, Seigel and Castellan (1988) provide a good overview. Briand et al. (1996) discuss the disadvantages of nonparametric statistics versus parametric statistics in software engineering; they point out that a certain amount of violation of the assumptions of parametric statistics is legitimate, but that nonparametric statistics should be used when there are extreme violations of those assumptions, as there may well be in field studies.

Qualitative analyzes do not rely on quantitative measures to describe the data. Rather, they provide a general characterization based on the researchers' coding schemes. For example, after interviewing software engineers at 12 organizations, one of us (Singer, 1998) found characteristics common to many of the organizations, such as their reliance on maintenance control systems to keep historical data. Again, the different types of qualitative analysis are too complex to detail in this paper. See Miles and Huberman, (1994) for a very good overview.

In summary, the way the data is coded will affect its interpretation and the possible courses for its evaluation. Therefore it is important to ensure that coding schemes reflect the research goals. They should tie in to particular research questions. Additionally, coding schemes should be devised with the analysis techniques in mind. Again, different schemes will lend themselves to different evaluative mechanisms. However, one way to overcome the limitations of any one technique is to look at the data using several different techniques (such as combining a qualitative and quantitative analyzes). A *triangulation* approach (Jick, 1979) will allow for a more accurate picture of the studied phenomena. Bratthall and Jørgensen (2002) give a very nice example of using multiple methods for data triangulation. Their example is framed in a software engineering context examining software evolution and development. In fact, many of the examples cited earlier, among them (Beecham et al., 2003; Budgen and Thomson, 2003; Seaman et al., 2003; Shull et al., 2000; Wu et al., 2003), use multiple methods to triangulate their results.

As a final note, with any type of analysis technique, it is generally useful to go back to the original participant population to discuss the findings. Participants can tell researchers whether they believe an accurate portrayal of their situation has been achieved. This, in turn, can let researchers know whether they used appropriate coding scheme and analysis techniques.

5. Conclusions

In this paper we have discussed issues that software engineering researchers need to consider when studying practitioners in the field. Field studies are one of several complementary approaches to software engineering research and are based on a recognition that software engineering is fundamentally a human activity: Field studies are particularly useful when one is gathering basic information to develop theories or understand practices.

The material presented in this paper will be useful to both the producer and consumer of software engineering research. Our goal is give researchers a perspective on how they might effectively collect data in the field—we believe that more such studies are needed. The material presented here will also help others evaluate published field studies: For example, readers of a field study may ask whether appropriate data gathering or analysis techniques were used.

In this paper, we divided the set of field study techniques into three main categories. First-degree techniques such as interviewing, brainstorming, and shadowing place the researcher in direct contact with participants. Second-degree techniques allow researchers to observe work without needing to communicate directly with participants. Third-degree techniques involve retrospective study of work artifacts such as source code, problem logs, or documentation. Each technique has advantages and disadvantages that we described in Section 2 and answers specific questions, as listed in Table 2.

To perform good field studies, a researcher must first create effective plans. The plans should describe the study techniques and also how various practical issues are to be handled. To choose study techniques, we espouse a modification of the GQM methodology, originally developed to choose metrics, but described here to choose data collection techniques. The researcher must have firm goals in mind, choose study questions that will help achieve the goals, and then choose one or more techniques that are best suited to answer the questions.

In addition to deciding which techniques to use, the researcher must also determine the level of detail of the data to be gathered. For most first degree techniques one must typically choose among, in increasing order of information volume and hence difficulty of analysis: manual notes, audio-taping and videotaping. In all three cases, a key difficulty is encoding the data so that it can be analyzed.

Regardless of the approach to gathering and analyzing data, field studies also raise many logistical concerns that should be dealt with in the initial plan. For example: How does one approach and establish relationships with companies and employees in order to obtain a suitable sample of participants? Will the research be considered ethical, considering that it involves human participants? And finally, will it be possible to find research staff who are competent and interested, given that most of the techniques described in this paper are labor intensive but not yet part of mainstream software engineering research?

Researchers wishing to learn about field studies in more depth can investigate literature on the topic in the social sciences. The purpose of this paper has been to raise awareness among software engineers of the options available—most software engineering researchers would not think of these techniques until their awareness is raised. However, there are some differences between the way social scientists would apply the techniques and the way software engineer researchers would apply them: The key difference is that the goal of software engineering researchers is to improve the software engineering process, as opposed to learning about social reality for its own sake. Secondly, software engineering researchers normally are software engineers as well, and are therefore part of the population they are studying—this impacts the depth of technical understanding those researchers can rapidly achieve, and the types of interactions they can have with study participants.

In conclusion, field studies provide empirical studies researchers with a unique perspective on software engineering. As such, we hope that others will pursue this approach. The techniques described in this paper are well worth considering to better understand how software engineering occurs, thereby aiding in the development of methods for improving software production.

Acknowledgements

This work is supported by NSERC and sponsored by the Consortium for Software Engineering Research (CSER).

Notes

1. Second degree contact is distinguished from third degree contact in that second degree contact requires data acquisition when work is occurring, while third degree contact requires only work artifacts and has no requirements with respect to when data is acquired.
2. However, please note that our observational data could be incomplete due to the Hawthorne effect, discussed earlier.
3. For example, we intended 'formal languages' to be the mathematical study of the principles of artificial languages in general, yet apparently some respondents thought we were referring to learning how to program.
4. It is often felt that videotaping will influence the participants actions. However, while videotaping appears to do so initially, the novelty wears off quickly (Jordan and Henderson, 1995).
5. System logging of the computer screen may provide an alternative in this situation, but it is still necessary to consider the video frame from the perspective of what data is required.

References

Anquetil, N., and Lethbridge, T. C. 1999. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice* 11: 201–221.

Baddoo, N., and Hall, T. 2002. Motivators of software process improvement: An analysis of practitioners' views. *Journal of Systems and Software* 62: 85–96.

Baddoo, N., and Hall, T. 2002. De-motivators of software process improvement: An analysis of practitioners' views. *Journal of Systems and Software* 66: 23–33.

Basili, V. R. Software modeling and measurement: The Goal/Question/Metric paradigm, Tech. Rep. CS-TR-2956, Department of Computer Science, University of Maryland, College Park, MD 20742, Sept. 1992.

Beecham, S., Hall, T., and Rainer, A. 2003. Software process improvement problems in twelve software companies: An empirical analysis. *Empirical Software Engineering* 8: 7–42.

Bellotti, V., and Bly, S. 1996. Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team. Cambridge, MA: Conference on Computer Supported Cooperative Work, pp. 209–219.

Berlin, L. M. 1993. Beyond Program Understanding: A Look at Programming Expertise in Industry. *Empirical Studies of Programmers*. Palo Alto: Fifth Workshop, 6–25.

Bratthall, L., and Jørgensen, M. 2002. "Can you trust a single data source exploratory software engineering case study?" *Empirical Software Engineering: An International Journal* 7(1): 9–26.

Briand, L., El Emam, K., and Morasca, S. 1996. On the application of measurement theory in software engineering. *Empirical Software Engineering* 1: 61–88.

Buckley, J., and Cahill, T. 1997. *Measuring Comprehension Behaviour Through System Monitoring*, Int. Workshop on Empirical Studies of Software Maintenance, Bari, Italy, 109–113.

Budgen, D., and Thomson, M. 2003. CASE tool Evaluation: Experiences from an empirical study. *Journal of Systems and Software* 67: 55–75.

Chi, M. 1997. Quantifying qualitative analyzes of verbal data: A practical guide. *The Journal of the Learning Sciences* 6(3): 271–315.

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): November, 1268–1287.

Damian, D., Zowghi, D., Vaidyanathasamy, L., and Pal, Y. 2004. An industrial case study of immediate benefits of requirements engineering process improvement at the australian center for unisys software. *Empirical Software Engineering: An International Journal* 9(1–2): 45–75.

Delbecq, A. L., Van de Ven, A. H., Gustafson, D. H. 1975. *Group Techniques for Program Planning. Scott*. Glenview, IL: Foresman & Co.

DeVaus, D. A. 1996. *Surveys in Social Research*. 4th edition. London: UCL Press.

Draper, S. 2004. The Hawthorne Effect. http://www.psy.gla.ac.uk/~steve/hawth.html.

Ericcson, K., and Simon, H. 1984. *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: The MIT Press.

Foddy, W. 1994. *Constructing Questions for Interviews and Questionnaires: Theory and Practice in Social Research*. Cambridge, MA: Cambridge University Press.

Hassan, A., Holt, R., and Mockus, A. 2004. MSR 20004: The international workshop on mining software repositories. *Proc. ICSE 2004: International Conference on Software Engineering*, Scotland, UK, May, pp. 23–28.

Herbsleb, J., and Mockus, A. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions Software Engineering* 29(6): 481–494.

Hungerford, B., Hevner, A., and Collins, R. 2004. Reviewing software diagrams: A cognitive study. *IEEE, Transactions Software Engineering* 30(2): 82–96.

Iivari, J. 1996. Why are CASE tools not used? *Communications of the ACM* 39(10): October, 94–103.

Jick, T. 1979. Mixing qualitative and quantitative methods: Triangulation in action. *Administrative Science Quarterly* 24(4): December, 602–611.

Jordan, B., and Henderson, A. 1995. Interaction analysis: Foundations and practice. *The Journal of the Learning Sciences* 4(1): 39–103.

Jørgensen, M. 1995. An empirical study of software maintenance tasks. *Software Maintenance: Research and Practice* 7: 27–48.

Keller, R., Schauer, R., Robitaille, S., and Page, P. 1999. Pattern-based reverse engineering of design components. In *Proc, Int. Conf. Software Engineering*, Los Angeles, CA, pp. 226–235.

Kemerer, C. F., and Slaughter, S. A. 1997. Determinants of software maintenance profiles: An empirical investigation. *Software Maintenance: Research and Practice* 9: 235–251.

Kensing, F. 1998. *Prompted Reflections: A Technique for Understanding Complex Work*. interactions, January/February, 7–15.

Lethbridge, T. C. 2000. Priorities for the education and training of software engineers. *Journal of Systems and Software* 53(1): 53–71.

Lethbridge, T. C., and Anquetil, N. 2000. Evaluation of approaches to clustering for program comprehension and remodularization. In H. Erdogmus and O. Tanir, (eds.), *Advances in Software Engineering: Topics in Evolution, Comprehension and Evaluation*, New York: Springer-Verlag.

Lethbridge, T. C., and Laganière, R. 2001. *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*. London: McGraw-Hill.

Miles, M. B. 1979. Qualitative data as an attractive nuisance: The problem of analysis. *Administrative Science Quarterly* 24(4): 590–601.

Miles, M. B., and Huberman, A. M. 1994. *Qualitative Data Analysis: An Expanded Sourcebook*. 2nd edition. Thousand Oaks, CA: Sage Publications.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. 2002. Two case studies of open source software development: Apache and mozilla. *ACM Trans. on Software Engineering and Methodology* 11(3) 209–246.

NASA. SEL COTS Study Phase 1 Initial Characterization Study Report, SEL-98-001, August 1998, http://sel.gsfc.nasa.gov/website/documents/online-doc.htm.

Nielsen, J. 1997. The Use and Misuse of Focus Groups. http://www.useit.com/papers/focusgroups.html.

Perry, D. E., Staudenmayer, N., and Votta, L. 1994. People, organizations, and process improvement. *IEEE Software* July, 37–45.

Pfleeger, S. L., and Hatton, L. 1997. Investigating the influence of formal methods. *Computer* February, 33–43.

Pfleeger, S., and Kitchenham, B. 2001. Principles of survey research Part 1: Turning lemons into lemonade. *Software Engineering Notes* 26(6) 16–18.

Porter, A. A., Siy, H. P., Toman, C. A., and Votta, L. G. 1997. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions Software Engineering* 23(6): 329–346.

Punter, T., Ciolkowski, M., Freimut, B., John, I. 2003. Conducting on-line surveys in software engineering. *Proceedings Int. Symp. on Empirical Software Eng. '03*, pp. 80–88.

Rainer, A., and Hall, T. 2003. A quantitative and qualitative analysis of factors affecting software processes. *Journal of Systems and Software* 66: 7–21.

Robbins, S. P. 1994. *Essentials of Organizational Behavior*. 4th edition. Englewood Cliffs, NJ: Prentice Hall.

Robillard, P. N., d'Astous, P., Détienne, D., and Visser, W. 1998. Measuring cognitive activities in software engineering. *Proc. 20th Int. Conf. Software Engineering*, Japan, pp. 292–300.

Sayyad-Shirabad, J., Lethbridge, T. C., and Lyon, S. 1997. A little knowledge can go a long way towards program understanding. *Proc. 5th Int. Workshop on Program Comprehension*. Dearborn, MI: IEEE, pp. 111–117.

Sayyad-Shirabad, J., Lethbridge, T. C., and Matwin, S. 2003. Applying data mining to software maintenance records. *Proc CASCON 2003*, Toronto, October, IBM, in ACM Digital Library, pp. 136–148.

Scacchi, W. 2003. Issues and experiences in modeling open source software processes. *Proc. 3rd. Workshop on Open Source Software Engineering*, Portland, OR: 25th. Int. Conf. Software Engineering, May.

Seaman, C. B., and Basili, V. R. 1998. Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering* 24(7): July, 559–572.

Seaman, C., Mendonca, M., Basili, V., and Kim, Y. 2003. User interface evaluation and empirically-based evolution of a prototype experience management tool. *IEEE Transactions on Software Engineering* 29: 838–850.

Seigel, S., and Castellan, N. J. 1988. *Nonparametric Statistics for the Behavioral Sciences*. 2nd edition. Boston, MA: McGraw-Hill.

Shull, F., Lanubile, F., and Basili, V. 2000. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering* 26: 1101–1118.

Sim S. E., and Holt, R. C. 1998. The ramp-up problem in software projects: A case study of how software immigrants naturalize. *Proc. 20th Int. Conf. on Software Engineering*, Kyoto, Japan, April, pp. 361–370.

Sim, S. E., Clarke, C. L. A., and Holt, R. C. 1998. Archetypal source code searches: A survey of software developers and maintainers. *Proc. Int. Workshop on Program Comprehension*, Ischia, Italy. pp. 180–187.

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

Singer, J. 1998. Practices of software maintenance. *Proc. Int. Conf. on Software Maintenance*. Washington, DC, November, pp. 139–145.

Singer, J., Lethbridge, T. C., and Vinson, N. 1998. Work practices as an alternative method to assist tool design in software engineering. *Proc. International Workshop on Program Comprehension*. Ischia, Italy, pp. 173–179.

Singer, J., and Vinson, N. 2002. Ethical issues in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 28: 1171–1180.

Snelling, L., and Bruce-Smith, D. 1997. The work mapping technique. Interactions 25–31, July/August.

Somé, S. S., and Lethbridge T. C. 1998. Parsing minimizing when extracting information from code in the presence of conditional compilation. *Proc. 6th IEEE International Workshop on Program Comprehension*. Italy, June pp. 118–125.

Teasley, S., Covi, L, Krishnan, M., and Olson, J. 2002. Rapid software development through team collocation. *IEEE Transactions on Software Engineering* 28: 671–683.

von Mayrhauser, A., and Vans, A. M. 1993. From program comprehension to tool requirements for an industrial environment. *Proc. of the 2nd Workshop on Program Comprehension*, Capri, Italy, July, pp. 78–86.

von Mayrhauser, A., and Vans, A. M. 1995. Program understanding: Models and experiments. In M. C. Yovita and M. V. Zelkowitz, (eds.), *Advances in Computers*, Vol. 40, Academic Press, pp. 1–38.

Walenstein, A. 2003. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. *Proc. the 11th IEEE Workshop on Program Comprehension*.

Walz, D. B., Elam, J. J., and Curtis, B. 1993. Inside a software design team: Knowledge acquisition, sharing, and integration. *Communications of the ACM* 36(10): October, 62–77.

Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R. 2000. Strengthening the case for pair-programming, *IEEE Software* July/Aug, 19–25.

Wolf, A., and Rosenblum, D. 1993. A study in software process data capture and analysis. *Proc. 2nd International Conference on Software Process* February, pp. 115–124.

Wu, J., Graham, T., Smith, P. 2003. A study of collaboration in software design. *Proc. Int. Symp. Empirical Software Eng.* '03.

Dr. Janice Singer currently heads the HCI research programme at the National Research Council Canada, a group of nine researchers investigating collaboration, privacy, and 3D navigation from a human-centred perspective. She is additionally a member of the Software Engineering Group. Dr. Singer's research interests include empirical software engineering, navigation in software spaces, collaborative software development, and research ethics from a software engineering perspective. Dr. Singer received her Ph.D. in Cognition and Learning from the Learning Research and Development Center of the University of Pittsburgh. Before coming to the NRC, she worked for Tektronix, IBM, and Xerox PARC.

Timothy C. Lethbridge is an Associate Professor at the University of Ottawa, Canada. His research applies empirical methods in the context of software tools for manipulating complex information, as well as software engineering education. His main research partner is currently IBM Ottawa. He is the author of a textbook on object-oriented software engineering, and helped develop SE2004, The IEEE/ACM guidelines for software engineering curricula.

Susan Elliott Sim is an Assistant Professor at University of California, Irvine. She has worked with a range of empirical methods, including experiments, case studies, ethnography, surveys, and benchmarks. She has conducted empirical studies at several software companies, including IBM. Sim is also a co-creator of GXL (Graph eXchange Language), an XML-based standard exchange format for software data. Her research interests include program comprehension, research methodology, and software process for small business.