Chapter 1
Software Engineering Data Collection for Field Studies[1]

Janice Singer, Susan E. Sim, and Timothy C. Lethbridge

Abstract. Software engineering is an intensely people-oriented activity, yet little is known about how 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 aid this goal, we describe a series of data collection techniques for such studies, organized around a taxonomy based on the degree to which interaction with software engineers is necessary. For each technique, we provide examples from the literature, an analysis of some of its advantages and disadvantages, and a discussion of special reporting requirements. We also talk briefly about recording options and data analysis.

1. Introduction

Software engineering involves real people working in real environments. People create software, people maintain software, people evolve software. Accordingly to understand software engineering, one should study software engineers as they work – typically by doing field studies. In this chapter, we introduce a set of data collection techniques suitable for performing such studies that can be used individually or in combination to understand different aspects of real world environments. These data collection techniques can be used with a wide variety of methods under a wide variety of theoretical and philosophical traditions (see Easterbrook et al., Chap. 11).

To better showcase the qualities of the various techniques, we have created a taxonomy based on the degree to which interaction with software engineers is required. The next section details the taxonomy. In Sect. 3, each technique is described in detail. We talk briefly in Sect. 4 about recording options for the data and present a brief overview of data analysis. We conclude the chapter with a discussion of how these techniques can be used in the most appropriate fashion.

[1] Based on Lethbridge, T., Sim, S., & Singer, J. (2005). Studying software engineers: data collection techniques for software field studies, *Empirical Software Engineering 10*(3), 311–341.

F. Shull et al. (eds.), *Guide to Advanced Empirical Software Engineering*.

2. Field Study Data Collection Taxonomy

Table 1. presents a summary of the data collection techniques; the second column shows the kinds of questions each can answer; the third column indicates the amount of data generated by the technique, and the fourth column shows other areas in software engineering where the technique is applied. Each technique is categorized according to how much contact is required between the researchers and the participants[2]. Direct techniques require the researcher to have direct involvement with the participant population. Indirect techniques require the researcher to have only indirect access to the participants' via direct access to their work environment. Finally, independent techniques require researchers to access only work artifacts, such as source code or documentation. Selecting an appropriate technique will be influenced by the questions asked and the amount of resources available to conduct the study. Generally, direct techniques require more resources, both to collect the data and to analyse it. Direct techniques are, however, the only techniques that allow researchers to explore the thoughts and feelings of the software engineers.

3. Survey of Data Collection Techniques

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 direct techniques, moving on to indirect techniques, and concluding with independent techniques. Each of the techniques is described in the same way. First the technique is described. Then its advantages and disadvantages are identified. Next, one or more examples of its use in software engineering research are given. Finally, some guidance is given regarding special considerations when reporting the technique (for more information on reporting in general, see Jedlitschka et al., Chap. 8).

3.1. Direct Techniques

The first five techniques listed in Table 1 are what we call *inquisitive* techniques (brainstorming, focus groups, interviews, questionnaires, conceptual modeling), while the remaining ones are primarily *observational*. Each type is appropriate for gathering different types of information from software engineers.

[2] We recognize that there is some debate about whether to properly characterize people who participate in research as subjects or participants. In this chapter, we have chosen to use the word participant because in field studies, there is frequently a greater degree of collaboration between those being studied and those doing the research.

Table 1 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
Direct 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
Interviews and questionnaires	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	Time sheets
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 to large	
Indirect techniques			
Instrumenting systems	Software usage over a long period, for many participants	Large	Software usage analysis
Fly on the wall	Time spent intermittently in one location, patterns of activities (particularly collaboration)	Medium	
Independent 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.

Inquisitive 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 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 (Draper, 2004; Robbins, 1994) 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. 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 uncover as many ideas as possible. Brainstorming works best with a moderator because the moderator can motivate the group and keep it focused. Additionally, brainstorming works best when there is a simple "trigger question" to be answered and everybody is given the chance to contribute their opinions. 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?" or "What features would you like to see in software engineering tools?" can result in extensive lists of valuable ideas that can then be analysed in more detail.

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. For more information on how to conduct focus groups, see Kontio et al., Chap. 4.

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 also very useful for collecting information (for instance about the usefulness of a particular tool) from large groups of people at once. 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 difficult 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 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.

Hall and her colleagues have published a number of papers based on a large study involving focus groups to understand software process improvement (see for example, Baddoo and Hall, 2002; Rainer and Hall, 2003). In their studies, 39 focus groups were implemented in 13 companies. The groups were comprised of between four and six participants. The companies were chosen based on certain characteristics, but overall were representative of the industry. Each session lasted 90 min. 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.

Storey et al. (2007) conducted a focus group with a number of users of a tool they developed. The focus group enabled the users to communicate with each other, and additionally allowed for greater time efficiency when collecting the data than interviews would have allowed.

Reporting guidelines: The reporting of brainstorming and focus groups is similar. For both, the number of participants seen, and the context in which they were seen should be reported. Where appropriate the role and expertise of the moderator should be described. If specific questions were used, they should be detailed. Additionally, the time spent on brainstorming or the focus group should be reported. Finally, the type of data recording used should be described (e.g., video, audio, notes, etc.).

3.1.2. Interviews

Interviews involve at least one researcher talking to at least one respondent. Interviews can be conducted in two ways. In a structured interview, a fixed list of carefully worded questions forms the basis of the interview. Usually, the questions

are asked exactly as written, and no deviations occur. The data from structured interviews is usually analysed using statistical analyses. In a semi-structured interview, the interview generally follows more of a conversational flow. New questions may be devised as new information is learned. Typically, some open-ended questions that allow for greater interaction are asked. Furthermore, in some semi-structured interviews, the interview will be structured around a framework of potential topics as opposed to any specific questions. The data from semi-structured interviews is usually analysed using qualitative analysis methods (see Seaman, Chap. 2).

Advantages: Structured interviews are an efficient means of collecting the same data from a large number of respondents. Semi-structured interviews tend to be much more 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 attend the meeting (whether in person, by phone, videoconference, or over the web). If the data from interviews consists of audio- or videotapes, this needs to be transcribed and/or coded; careful note-taking may, however, often be an adequate substitute for audio or video recording. Finally, participants' reports of events may not mirror reality. For instance, in one of our interview studies, developers reported that they spent a substantial amount of time reading documentation, but we did not observe this to be true.

Examples: Interviews have been used in many studies because they fit well with many types of methods and philosophical traditions. 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 interviews were semi-structured with open-ended questions.

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, understanding of the market need, and understanding of requirements.
Reporting guidelines: When reporting data from interviews, it is necessary to detail the number and type of interviewees seen, approximately how long the interviews took, the type of interview (semi-structured or structured), the way the interview is recorded, and how the participants were selected. Additionally, if possible, provide a copy of the questions in the report or an appendix.

3.1.3. Questionnaires

Questionnaires are sets of questions administered in a written format. These are the most common field technique 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) (see also Chap. 3). This series gives detailed information about how to design and implement questionnaires. Punter et al. (2003) further provide 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, response 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. 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, then low response rates may 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. Finally, as with questionnaires, developers' responses to questions may not mirror reality.
Examples: Lethbridge (2000) used questionnaires that were partly web-based and partly paper-based to learn what knowledge software engineers apply in their daily work, and how this relates to what they were taught in their formal education.

Respondents were asked four questions about each of a long list of topics. Several questionnaires were piloted, but nonetheless a couple of the topics[3] were interpreted in different ways by different respondents. Despite this, useful conclusions about how software engineers should be trained were drawn from the study.

Iivari (1996) used a paper-based questionnaire to test nine hypotheses about factors affecting CASE tool adoption in 52 organizations in Finland. The author contacted organizations that 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.

Reporting guidelines: When reporting data from questionnaires, it is necessary to detail how the population was sampled (i.e., who the questionnaires were sent to, or how respondents were chosen) and the response rate for the questionnaire, if appropriate. Any piloting and subsequent modification of the questionnaire should be explained. Additionally, if possible, provide a copy of the questions in the report or an appendix.

3.1.4. 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.

Advantages: Conceptual models 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 frequently 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: 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

[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.

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 conceptual modeling 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 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 allowed 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.

Reporting guidelines: The most important thing to report for conceptual models is the exact instructions given to the participants and a precise description of the tools that they had available to them to model. The way the data is recorded should also be outlined.

3.1.5. 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. A special form of the work diary is time sheets. Many software engineers (particularly consultants) are required to maintain and update quite detailed time sheets recording how many hours are spent per day per activity category. These time sheets can be a valuable source of data.

If you are considering utilizing prompted work diaries, Karahasanovic et al. (2007) provide a comprehensive comparison of this technique to think-aloud protocol analysis (detailed below), evaluating its costs, impacts on problem solving, and benefits.

Advantages: Work diaries can provide better self-reports of events because they record activities on an ongoing basis rather than in retrospect (as in interviews and questionnaires). 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 may have significant problems with accuracy. Another problem with work diaries is that they may 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 1 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, and that communication patterns vary widely amongst developers. As a slight variation, at the end of each day, Izquierdo et al. (2007) asked developers to complete a communication diary that detailed who they talked to and the purpose for the communication. These were used as the basis to create social networks for the group.

As another example, Jørgensen (1995) randomly selected software maintainers and asked them to complete a form to describe their next task. 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.

In an interesting application the use of time sheets as data, Anda et al. (2005) describe a project where Simula Research Laboratory acted as both clients and researchers in an IT project, where the actual contract was given to four different companies, which allowed for a comparative case study. Although the applicability of this model in empirical software engineering is limited (because of the large amount of resources required), the paper nonetheless highlights how this data can potentially be used in a study (when collected from accessible sources).

Reporting guidelines: When reporting work diaries, the precise task given to the software engineers (e.g., to record their communication patterns) must be described, as well as how it was accomplished (e.g., reported to experimenter, recorded periodically throughout the day, etc). Additionally, the tools made available to do so should be detailed.

3.1.6. 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 2 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). Additionally, if you are considering utilizing this technique, Karahasanovic et al. (2007) provide a comprehensive comparison of this technique to a form of work diaries, evaluating its costs, impacts on problem solving, and benefits.

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 eliminating 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 technique translates to other domains where it is impossible to map out the problem space a priori. However, Chi (1997) has defined a technique called Verbal Analysis that does address this problem. In either case, 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 human cognitive processes are represented by the contents of short-term memory that are 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.
Reporting guidelines: When reporting think-aloud protocol analysis, it is important to provide an extremely precise characterization of the task the participant was asked to undertake, including any tools at the participant's disposal. The time taken to complete the task and any materials provided to the participant are also important to describe. Finally, the precise way in which the analysis occurs needs to be closely detailed, especially if it is based on information processing theory or a specific cognitive model.

3.1.7. 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 difficult 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 (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 used *synchronized shadowing* where two experimenters used two laptop computers to record the software engineer's actions. One 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.

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. Similarly, Ko et al. (2007) also shadowed software engineers. They asked the participants to think of the researchers as a new hire to which they should explain what they were doing. From this data, they were able to categorize the met and unmet information needs of software engineers.

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.

Reporting guidelines: In reporting shadowing, the precise form of shadowing and/or observation needs to be detailed, including whether any verbal instructions were given to the participant to think out loud. Additionally, the way the information is recorded must be detailed as well as the length of the session, and any other special instructions given to the participants. It is also helpful to provide context information, such as what activities the shadowed and/or observed participants were engaged in, and whether this was typical or not.

3.1.8. Participant-Observer (Joining the Team)

Usually done as part of an ethnography, in the Participant-Observer technique, 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 techniques used by Seaman and Basili (1998) in their studies of how communication and organization factors affect the quality of software inspections. 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. (1997) also used the participant-observer technique. 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.

More recently, Izquierdo et al. (2007) joined a team over a period of 4 months to understand how they processed information and became aware of changes. Izquierdo did not participate in any development, but rather used the opportunity of closeness to support data collection and a greater comprehension of the team dynamics.

Reporting guidelines: Using the participant-observer technique, it is important to report the role of the participant-observer in the team – whether they are actually involved in any of the meaningful team activities or not. It is also important to characterize how they interact with the team, and what access they have to team material. Additionally, the length of time of the interaction needs to be reported. Finally, a characterization of how data was collected, coded, and analysed must be provided.

3.2. Indirect Techniques

Indirect techniques require the researcher to have access to the software engineer's environment as they are working. However, the techniques do not require *direct* contact between the participant and researcher. Instead data collection is initiated, then the software engineers go about their normal work as the data is 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). Related to this, Johnson and his group have developed Hackystat, an open-source server-based system for monitoring actions. Developers install sensors on their machines that then relay information to a centralized server (see www.csdl.ics.hawaii.edu/Research/hackystat for more information).

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 technique 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. 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.

More recently, Storey et al. (2007) logged developers' use of their TagSEA tool. The logs were stored on the client machine. The software engineers downloaded them to a server at specified intervals. The logs enabled Storey et al. (2007) to understand how the tool was being used, and nicely complemented other data sources such as interviews and a focus group. Similar to this study, Zou and Godfrey (2007) used a logger to determine which artifacts software maintainers were just viewing, and which were actually changed.
Reporting guidelines: The precise nature of the logging needs to be reported, including any special instrumentation installed on the software engineer's machines. This should include a description of what exactly is logged, with what frequency. Any special considerations with respect to data processing and analysis should also be detailed.

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 technique 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 (1993) asked mentors and apprentices at a software organization to audiotape their meetings in order to study how expertise is passed on. She later analyzed these recordings for patterns in conversations. She found that discussions were highly interactive in nature, using techniques such as confirmation and restatement to verify messages. Mentors not only explain features of the system; they also provide design rationale.

Walz et al. (1993) had software engineers videotape team meetings during the design phase of a development project. 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 keep track of 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
Reporting guidelines: The precise nature of the recording needs to be reported, along with any special instructions given to the participants. Additionally, any problems with the recording need to be reported, such as developers forgetting to record a meeting. Context information will also help to clarify the application of the technique, such as where the recording occurred, what the typical tasks were, who was involved, who was responsible for the recording, etc. Additionally, any methods used to transform, transcribe, and analyse the data need to be specified.

3.3. Independent Techniques

Independent 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.

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 issue tracker, problem reporting, change request and configuration management systems. These systems require software engineers to input data, such as a description of a problem encountered, or a comment when checking in a source code module. The copious records generated for such systems are a rich source of information for software engineering researchers. Besides the examples provided below, see the proceedings from the International Workshops on Mining Software Repositories.

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 (1997) analyzed reports of faults in an air traffic control system to evaluate the effect of adding formal methods to the development process. 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.

Researchers at NASA (1998) studied data from various projects in their studies of how 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.

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.

Reporting guidelines: The exact nature of the collected data needs to be specified, along with any special considerations, such as whether any data is missing, or uninterpretable for some reason. Additionally, any special processing of the data needs to be reported, such as if only a certain proportion is chosen to be analysed.

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 source code control systems. 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 independent 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 the proceedings from the International Workshops 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.

Reporting guidelines: As with instrumentation, the exact nature of what is being collected needs to specified, along with any special concerns, such as missing data. Additionally, if the data is processed in any way, it needs to be explained.

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.
Reporting guidelines: The documentation needs to be described as well as any processing on it.

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.
Reporting guidelines: The documents (e.g. source code) that provide the basis for the analysis should be carefully described. The nature of the processing on the data also needs to be detailed. Additionally, any special processing considerations should be described.

4. Applying the Techniques

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 outline how to record and analyze the data.

4.1. Record-Keeping Options

Direct techniques generally involve 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. 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. Related to videotaping, there are a number of software programs that allow screen capture and playback of the recorded interactions. To be used with videotape, the video and the screen capture must be synchronized in some way.

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

[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).

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. Once coded, the data is usually coded by another researcher to ensure the validity of the rating scheme. This is called inter-coder or inter-rater reliability. There are a number of statistics that can be reported that assess this, the most common is Kendall's tau.

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. 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.

Both quantitative and qualitative analysis can be supported by software tools. The most popular tools for quantitative analysis are SAS and SPSS. A number of different tools exist for helping with qualitative analysis, including NVivo, Altas/ti, and Noldus observer. Some of these tools also help with analysis of video recordings.

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, 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 chapter 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 chapter will be useful to both producers and consumers of software engineering research. Our goal is to give researchers a perspective on how they might effectively collect data in the field – we believe that more studies like this are needed. As well, the reporting guidelines presented here will 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 chapter, we divided the set of field study techniques into three main categories. Direct techniques such as interviewing, brainstorming, and shadowing place the researcher in direct contact with participants. Indirect techniques allow researchers to observe work without needing to communicate directly with participants. Independent 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 Sect. 2.

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 direct 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?

Finally, as technology and knowledge evolve, new data collection techniques emerge – e.g., using web cameras to collect work diaries. A good place to learn more about these new techniques is by following the human computer interaction and psychology methods literature. As well, reading papers in empirical software engineering will highlight current accepted techniques in the field, and how they may be used in practice.

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 and theories for improving software production.

References

Anda, B., Benestad, H., and Hove, S. 2005. A multiple-case study of effort estimation based on use case points. In *ISESE 2005* (Fourth International Symposium on Empirical Software Engineering). IEEE Computer Society, Noosa, Australia, November 17–18, pp. 407–416.

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

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

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

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

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.

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

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

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), 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., and Gustafson, D.H. 1975. *Group Techniques for Program Planning*. Scott, Foresman & Co, Glenview, IL.

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*. The MIT Press, Cambridge, MA.

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

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

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

Izquierdo, L., Damian, D., Singer, J., and Kwan, I. (2007). *Awareness in the Wild: Why Communication Breakdowns Occur*. ICGSE 2007, Germany.

Jick, T. 1979. Mixing qualitative and quantitative methods: triangulation in action. *Administrative Science Quarterly*, 24(4), 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.

Karahasanovic, A., Hinkel, U., Sjøberg D., and Thomas, R. (2007). *Comparing of Feedback Collection and Think-Aloud Methods in Program Comprehension Studies*. Accepted for publication in Journal of Behaviour & Information Technology, 2007.

Keller, R., Schauer, R. Robitaille, S., and Page, P. 1999. Pattern-based Reverse Engineering of Design Components. *Proceedings, International Conference on Software Engineering*, Los Angeles, CA, pp. 226–235.

Ko, A.J., DeLine, R., and Venolia, G. (2007). Information needs in collocated software development teams. *International Conference on Software Engineering (ICSE)*, May 20–26, 344–353.

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

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*, Second Edition. Sage Publications, Thousand Oaks, CA.

Mockus, A., Fielding, R.T., and Herbsleb, J.D. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions 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.

Perry, D.E., Staudenmayer, N., and Votta, L. 1994. People, organizations, and process improvement. *IEEE Software*, 11, 37–45.

Pfleeger, S.L. and Hatton, L. 1997. Investigating the influence of formal methods. *Computer*, 30, 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 on Software Engineering*, 23(6), 329–346.

Punter, T., Ciolkowski, M., Freimut, B., and John, I. 2003. Conducting On-Line Surveys in Software Engineering. *Proceedings on the International Symposium on Empirical Software Engineering'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*, Fourth edition. Prentice Hall, Englewood Cliffs, NJ.

Robillard, P.N., d'Astous, P., Détienne, D., and Visser, W. 1998. Measuring Cognitive Activities in Software Engineering. *Proceedings on the 20th International Conference on 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. *Proceedings of 5th International Workshop on Program Comprehension*, IEEE, Dearborn, MI, pp. 111–117.

Seigel, S. and Castellan, N.J. 1988. *Nonparametric Statistics for the Behavioral Sciences*, Second Edition. McGraw-Hill, New York.

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), 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.

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. *Proceedings on the 20th International Conference on Software Engineering*, Kyoto, Japan, April, pp. 361–370.

Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. 1997. An Examination of Software Engineering Work Practices. *Proceedings of CASCON*, IBM Toronto, October, pp. 209–223.

Somé, S.S. and Lethbridge T.C. 1998. Parsing Minimizing when Extracting Information from Code in the Presence of Conditional Compilation. *Proceedings of the 6th IEEE International Workshop on Program Comprehension*, Italy, June, pp. 118–125.

Storey, M.-A., Cheng, L., Singer, J., Muller, M., Ryall, J., and Myers, D. (2007). *Turning Tags into Waypoints for Code Navigation*. ICSM, Paris, France.

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. *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, 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, New York, pp. 1–38.

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), 62–77.

Walenstein, A. 2003. Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering. *Proceedings of the 11th IEEE Workshop on Program Comprehension*.

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. *Proceedings of the 2nd International Conference on Software Process*, February, pp. 115–124.

Wu, J., Graham, T., and Smith, P. 2003. A Study of Collaboration in Software Design. *Proceedings of the International Symposium on Empirical Software Engineering'03*.

Zou, L. and Godfrey, M. 2007. An Industrial Case Study of Program Artifacts Viewed During Maintenance Tasks. *Proceedings of the 2006 Working Conference on Reverse Engineering (WCRE-06)*, 23–28 October, Benevento, Italy.