Observations On Using Empirical Studies in Developing a Knowledge-Based Software Engineering Tool

David F. Redmiles

Department of Computer Science and Institute of Cognitive Science
University of Colorado
Boulder, Colorado 80309-0430
redmiles@cs.colorado.edu
(303) 492-1503

Abstract

There exist a wide variety of techniques for performing empirical studies which researchers in human-computer interaction have adapted from fields of cognitive psychology, sociology, and anthropology. An analysis of several of these techniques is presented through an approach that balances empirical study with tool development. The analysis is based on, and illustrated with, a several year experience consulting in a scientific software environment and in building and evaluating a prototype, knowledge-based tool to capture aspects of that experience. Guidelines for applying specific techniques and cautions about potential pitfalls are discussed. Many additional examples of using the techniques are cited from the literature.

1. Introduction

There exist a wide variety of techniques for performing empirical studies which researchers in human-computer interaction have adapted from fields of cognitive psychology, sociology, and anthropology. These techniques have been applied to the study of programmers (e.g., see [34]) and to the study of the roles and interaction of participants in large software development projects (e.g., see [6]). However, these techniques are rarely applied to the development and evaluation of software tools for augmenting human performance.

There are several potential obstacles to controlled empirical studies of software tools. First, such studies require an integration of different research cultures: cognitive psychology, sociology, anthropology, and software engineering. Second, the effectiveness of software tools is often difficult to measure statistically due to the phenomenon that programming is an activity that greatly exaggerates individual differences [7]. Third, development of tools is often driven by market considerations or by a few users selected for alpha testing. ''Evaluation'' tends to focus on the accuracy with which ''features'' have been implemented.

In an attempt to bridge cultural research boundaries, several techniques for empirical study are adopted into a framework for developing and evaluating knowledge-based, software engineering tools. The approach is based on, and illustrated with, a several year experience consulting in a scientific software environment and in building and evaluating a prototype, knowledge-based tool called EXPLAINER to capture aspects of that experience. Furthermore, a way to measure a tool's effectiveness amidst the phenomenon of individual differences is discussed. As the case study of EXPLAINER would in a sense provide only one data point for making observations about evaluation, this paper draws heavily on the literature to substantiate the use and usefulness of specific techniques.

The paper is organized as follows. First a description is given of the development of the EXPLAINER tool as a case study for use throughout the paper. Then, a general framework of tool development and empirical study is discussed. This is followed by a description of specific empirical techniques and potential pitfalls in applying them. The paper ends with a discussion of related work and conclusions.

2. A case study: development of the EXPLAINER tool

The development of the EXPLAINER tool is described in order to provide a context for discussing the interplay of experience, empirical study, and tool development, that is referred to throughout the paper. Additional sources of information about EXPLAINER are cited as appropriate and in general may be found in [27], [19], and [28].

The EXPLAINER prototype tool shown in Figure 1 is

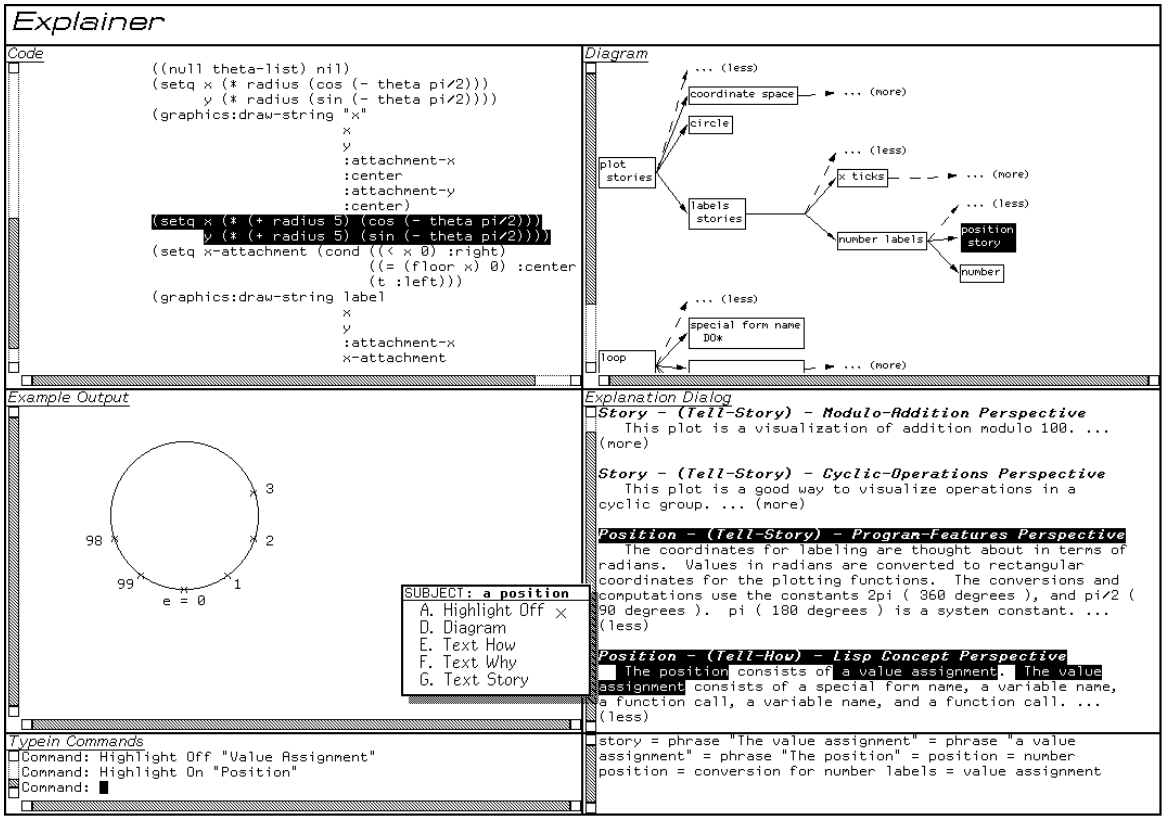

Figure 1: Explainer's Presentation of a Program Example

The EXPLAINER tool emulates some aspects of a human consultant, helping users of software libraries understand components in the context of complete program examples. The tool developed from observations during a several year work experience and through different stages of empirical evaluation. Different views provide multiple explanations of features illustrated by an example. Correspondences between an example's features and its explanation support software developers solving new tasks by analogy to the explained example.

intended to help people use software library components by explaining the components in the context of complete examples. Although EXPLAINER was implemented for the domain of programming components for computer graphics, it is envisioned that higher level specifications or other artifacts could be the object of explanation.

The value of examples became most apparent to me while consulting for users of a FORTRAN graphics library called DISSPLA [12]. The documentation for the DISSPLA library was organized into sections of features: one section on basic plotting, another section on using fonts, another on contour plots, etc. At the end of each section was a series of plots and FORTRAN code for calling the library subroutines that created the plot. Eventually, I found that by using the examples section, I could advise users about features I had never used myself. I would listen to their description of how they wanted to plot some data, then look through the examples for something similar. They could look at the pictures and concur or make other suggestions. With the given code, I could identify what subroutines produced the feature they wanted and extrapolate from the context of the example what parameters might be needed. When it was not obvious from the example what a parameter meant, a general explanation could be sought for in the earlier text part of the documentation section.

Over a time of about four years, this experience led implicitly to a conceptual framework, or process, of consulting by listening to a user's problem description, retrieving a related example, and, from the example, isolating functions and descriptions supporting a desired feature. This process is diagramed in Figure 2. This experience was supported by findings in the cognitive psychology literature on programming and problem solving (e.g., see [25], [16]) and in the computer science literature on automatic planning, especially case-based reasoning (e.g., see [1], [13]). After identifying a first approximation to this conceptual framework, programmers in a consulting situation could be observed for more detailed information, in particular, the kind of knowledge they needed to support their exploration of examples and solving of new problems based on the examples. It was

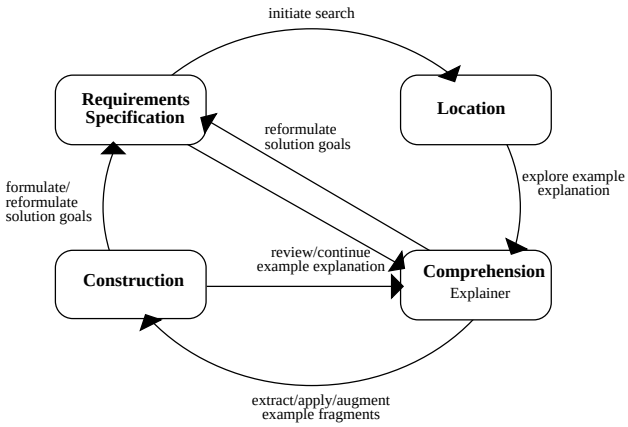

Figure 2: An Example-Based Software Development Process

The EXPLAINER tool co-evolved in a larger, conceptual framework for example-based software design. From initial requirements specification, software examples are located. The EXPLAINER tool helps users comprehend relevant parts of the examples enabling the users to identify whether parts of the example can be reused, whether new features should be added to the requirements, and/or whether additional examples should be sought.

observed that these questions focused on different perspectives of knowledge depending upon the problem-solving context [26].

A prototype tool was then developed based on a refinement of the conceptual framework and the more precise understanding of the knowledge involved [9]. This prototype was tested informally in a class on LISP programming as reported at this conference in 1991 [19]. Feedback from this testing identified several aspects of the interface that had to be changed (menu actions, highlighting) and additional types of knowledge that had to be added (e.g., knowledge about program execution).

A revised prototype was then tested more rigorously to ‘‘prove’’ the approach. Attending to the insights and problems identified in the previous study enabled this formal study to proceed without unexpected breakdowns (either of the tool itself or of the kinds of knowledge users expected). Refinements in the conceptual framework allowed conditions for this second study to be more easily identified. The obstacle of individual differences in programming that led to great variability in other tests of programmers became the focus for comparing the conditions. A group of test users working with the EXPLAINER tool could be identified as being helped more than users in other conditions in that the EXPLAINER group exhibited significantly less variation while still attaining good performance (see [28] for details).

3. Balancing empirical study with tool development

The preceding chronology of the development of the EXPLAINER tool illustrates four phases:

1. *Conceptual.* A conceptual framework or understanding of a methodology is gained from long-term observation.
2. *Observational.* More focused observations of potential users are employed to verify and enhance aspects of this conceptual framework, including specific kinds of knowledge a tool would need to support.
3. *Alpha.* An initial prototype is developed, evaluated with users, and revised.
4. *Beta.* The revised prototype is tested under controlled conditions to provide certification of the conceptual framework and the tool that embodies it.

This mutually supportive interrelationship among tools, conceptual frameworks, and empirical study is beginning to be more generally recognized: ‘‘systems are informing theory just as theory can inform the design of systems [22, p. 4].’’ (See also [2]).

For this paper, the terms ‘‘theory’’ or conceptual framework and ‘‘empirical study’’ will be generalized from their common usage in cognitive psychology. A theory or conceptual framework can exist explicitly, though informally, in a methodology, such as the diagram in Figure 2; explicitly and formally in a mathematical model, such as Anderson’s ACT* models in [25] and Kintsch’s construction-integration model in [20, 23]; or implicitly in the culture of a specific tool-based environment, such as UNIX or INTERLISP [33]. The notion of empirical study will include not only formally structured observations but also unstructured observations that can be made in the course of experience.

Table 1 summarizes potential activities during different phases of development. Constructing a table such as this is somewhat dangerous; there are several alternatives for filling in different activity cells for example techniques. In particular, almost all of the example techniques mentioned could be used in each phase. For example, Curtis and colleagues reported protocols and performed surveys in developing a conceptual framework at a social and organizational level [6, 37]. However, the point of this table is pedagogical, highlighting what can be done and what was successful in the EXPLAINER experience. In sections that follow, details of different techniques will be provided to help clarify the applicability of techniques in different circumstances and potential hazards in applying them.

Table 1: Role of Empirical Study in Development

Phase	Focus	Example Techniques	EXPLAINER Experience
1. Conceptual	conceptual framework	anecdotal	software library consulting
2. Observational	conceptual framework, hypothetical tool	protocol	question-answer dialogs
3. Alpha	tool conceptual framework	protocol, survey, statistical, cognitive walkthrough	first prototype in class
4. Beta	tool	statistical, survey, protocol	revised prototype among sample users

4. Specific techniques

As noted above, different empirical study techniques can be applied in many situations. Five characteristics are chosen to help delineate applicability of different techniques depending on the goals and circumstances of the study: length of time, number of observations, number of conditions, formality of analysis, and setting. Table 2 may be used to help identify the conditions under which these techniques are frequently applied. (The term ''unstructured'' is used instead of ''real-world'' or ''large-scale'': studies in normal work settings may be structured by the observer's own conceptual framework. Similarly, ''structured'' is used instead of ''laboratory'': informal observations with many variable conditions may also take place in an experimenter's chosen setting.) Each of the techniques is discussed in turn.

Anecdotal. Anecdotes are recollections of exceptional events. The events can be exceptional successes or failures. They shape our experiences and, when generalized and reinforced, can shape a methodology or conceptual framework. They develop and collect over a long period of time without the bias of a conceptual framework and, in the case of exceptional failures, can challenge preconceptions. In the EXPLAINER case study, an exceptionally successful consulting event occurred when, by simply examining examples of contour plots, I was able to explain to a user which subroutines to use in altering the labeling of the contours. This exceptional success punctuated the notion that examples supported the use of software features not learned a priori.

Anecdotes are becoming a focus of research. In the field of case-based reasoning, researchers use generalization (and other processes) from previous episodes or cases as as basis for a model of formal reasoning [1, 13]. Schank proposes that stories form a primary basis for our understanding and communication [30]. Stories are extrapolated into general issues in the field of design rationale and argumentation [21]. Petroski proposes that exceptional failures lead to improvements in the theory of design [24].

A potential hazard in relying solely on anecdotes is the fact that they are recollections and are therefore subject to biases of memory. A safeguard for this is built into the observational phase as discussed above; generalizations from anecdotes can be more objectively verified through observation, e.g., using protocols.

Protocol. Protocols are transcripts of people's words and actions. They can be applied in many situations. In the observational phase of EXPLAINER, verbal protocols were used to help analyze questions software users asked a consultant. Some example user statements include ''What is the syntax for the DO function?'' and ''I didn't know what this part [of the code] did.'' Collectively analyzing the verbal protocols provided evidence for the types of knowledge that would be needed in EXPLAINER such as knowledge perspectives about LISP and sample program execution. The protocols have the benefit of being more objective than anecdotes. Unlike personal recollections, they capture raw data that may be inspected by multiple researchers. Furthermore, they remain constant while a conceptual framework is refined. Thus, time permitting, the same protocols can be re-examined from different points of view.

Verbatims have been a major tool in the fields of linguistics and cognitive psychology. Recent work has studied the use of ''thinking-aloud'' protocols to objectively access peoples' mental conceptualizations during problem solving [8, 15]. The availability of video now makes possible the observation of peoples' attention in a study [36]. For example, in studying software tools, the protocol technique allows careful observation of which of

Table 2: Common Characteristics of Techniques

Technique	Length of Time	Number of Observations	Number of Conditions	Formality of Analysis	Setting
Anecdotal	years, months	many	many	low	unstructured
Protocol	hours days	few	few	low	unstructured
Cognitive Walkthrough	minutes	few	few	high	structured
Statistical	minutes, hours	many	few	high	structured
Survey	minutes	many	few	low	structured

information and views users focus attention on when solving a task.

A major limitation of protocols is the amount of data and the time to analyze it collectively. A collective analysis is critical to avoid misinterpretation of peoples' intent. In long-term studies, the amount of video or audio footage would be preventive. Also, studies that involve the cooperation of many participants working in different locations are often preventive even in the short-term due to the amount of equipment required, though some corporations have committed resources to efforts in ubiquitous computing that make such studies feasible [38].

Cognitive Walkthrough. In a cognitive walkthrough, a designer projects him or herself into the role of a potential end user and attempts to carry out a task anticipated by the tool. The walkthrough is controlled by answering specific questions relating task and interface, assuming a specific user background. The primary focus of the questions is whether the user will recognize easily in the interface, the way (e.g., menu item) to perform a step necessary to complete a task. In the EXPLAINER development, a cognitive walkthrough was performed before the beta test. A simple task was evaluated, that of identifying and highlighting in a program example, the piece of code that labeled an axis (see Figure 3). Though the menu included features users had claimed to need, its combined complexity (length, organization, choice of terms) was such that it failed the walkthrough analysis. This led to the simplified menu seen in Figure 1.

Cognitive walkthroughs are a new technique in the field of human-computer interaction. While technical walkthroughs [10] seek to verify if the functionality of a tool is implemented accurately; cognitive walkthroughs [17] seek to verify if a potential end user will find and understand a tool's intended functionality. In a sense, cognitive walkthroughs are an attempt to simulate a protocol with a real user, in particular, to identify potential breakdowns. They have the advantage that they can be performed without the overhead of involving actual subjects.

An obvious potential for failure of a walkthrough is the bias of the designer evaluating his or her own tool. The formal questions used in a walkthrough have also been incorporated into prompting tools. These discipline the evaluator in a way to encourage objectivity. Objectivity can be further improved if the designer performing the walkthrough did not select the task. Internal studies performed by Lewis and colleagues at the University of Colorado attempt to refine the walkthrough method, e.g., by involving teams of evaluators. For more than simple trial tasks however, walkthroughs produce large quantities of data to be analyzed.

Statistical. Statistical techniques are the most formal of those mentioned so far and the most common technique involves tests of hypotheses (e.g., see [14]). In the final (beta) evaluation of EXPLAINER, tests of hypothesis were employed to demonstrate that users of the EXPLAINER tool could solve a sample programming task better than users working under other conditions. To directly employ a test of hypothesis, the difference between groups of users had to be limited to one variable. In the EXPLAINER case, the one variable selected was the tool: one group used the EXPLAINER tool to understand a program example and adapt it to a new task, while a second group used a commercial, on-line systems documentation tool to understand the program example. A more detailed account of the evaluation is found in [28] and [27]. It is worth noting that the author's initial instinct was to vary two variables: both the tool and the use of an example. However this would have led to an ambiguity of which factor contributed to success or failure. Furthermore, if the value of the example-based framework was really believed, the first group would have had an unfair advantage.

Texts treating the design of experiments and procedures for test of hypotheses are common (e.g., see

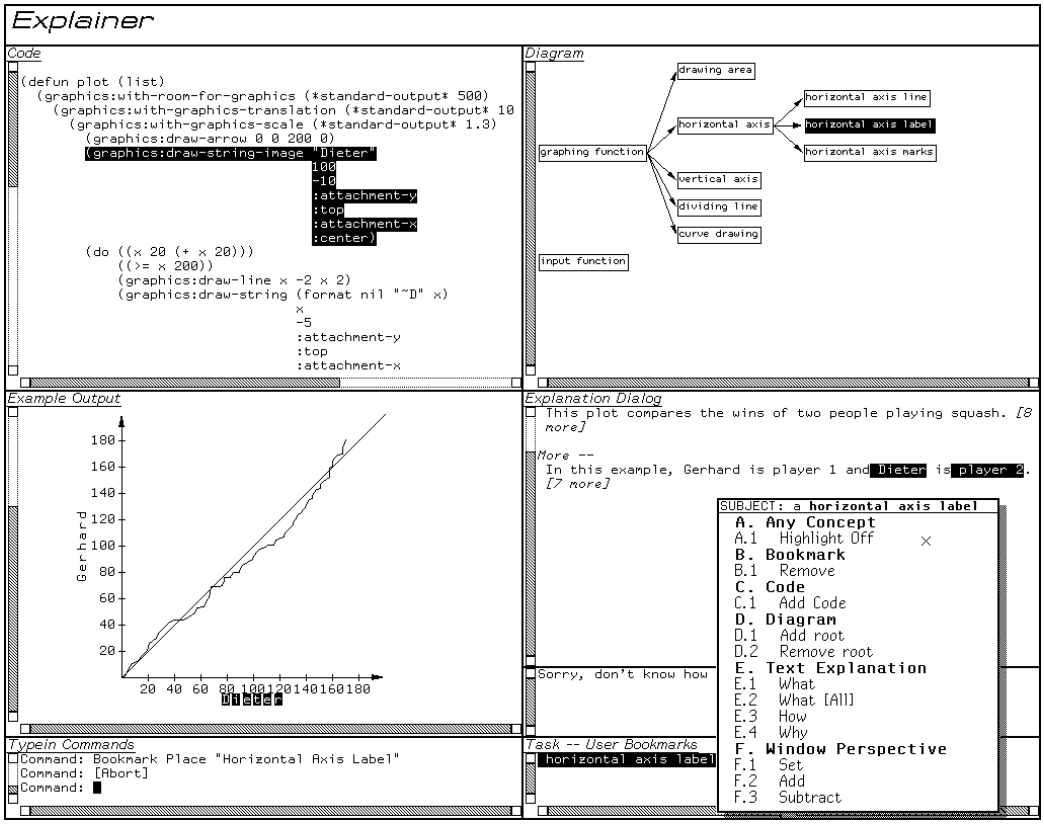

Figure 3: Explainer Before Cognitve Walkthrough

One version of the EXPLAINER tool was revised without the need for testing ''live'' users. As described in the text, the cognitive walkthrough technique revealed that users would not be able to use the menu shown in the lower right of the screen to locate the functionality they needed to solve common tasks EXPLAINER was intended to support. The menu was revised as it appears in Figure 1.

[18]). However, the application of these techniques to the evaluation of programming tools is not (at least from the perspective of tools that are intended to augment human performance). The major obstacle is that the effectiveness is often difficult to measure statistically due to the phenomenon that programming is an activity that greatly exaggerates individual differences [7]. Specifically, effects of tools are often obscured by large variances in measured variables. In the EXPLAINER evaluation, less common statistical tests were employed to circumvent this problem. First, non-parametric methods were employed for comparing medians [35]. These techniques could be adopted to accommodate measurements other than continuous real variables. For example, since these techniques often employ ranks of measurements, they could be adapted to include the ''infinitely bad'' performance by a subject who quit. These values are often discarded as ''null'' observations. Second, techniques focusing on the variance of different groups were also pursued. These led to the analysis in [28] that the effectiveness of the EXPLAINER tool could in fact be observed in the reduction of the variance (while performance measures remained constant). A more general lesson learned was that an honest, objective look at the data and what it implied can lead to a deeper understanding of the cognitive processes of users of tools; adherence to an evaluation methodology can force an experimenter away from preconceived conclusions.

Survey. Surveys are a frequent method of determining people's opinions in all matters, including software tools. A survey was used in the EXPLAINER evaluation to solicit information about what tool features users felt were particularly useful and which were not. For example, through the survey, users commented frequently that the highlighting command that showed correspondences between information in the different EXPLAINER views (e.g., showing the correspondence between a text sentence, a function call, and a diagram node) was one of the most useful features.

As has been pointed out in other studies (e.g., see [6]), and in general (e.g., see [11]), surveys are often subjective. In the EXPLAINER evaluation, an attempt was made to counter the bias of subjects by working through the survey with them. The experiment leader was able to

question user responses that contradicted their actions. For example, when one subject was asked to rate the usefulness of the diagram view (upper right of Figure 1), he responded at first with a low rating. When reminded that he worked with it during the test, expanding nodes about label drawing, he realized that the diagram view had actually helped him solve part of the task. Survey data is subjective, but learning users intentions through surveys and interviews can be helpful in the interpretation of the objective data.

5. Related work

Survey articles reveal a vast literature in the psychology of programming [4, 5, 32]. The contributions range from understanding cognitive effects (e.g., see [34], [3], [25]) to social and organizational effects (e.g., see [31], [6]). However, few instances of studies that perform controlled comparison of conditions can be found that evaluate software tools for improving performance. This paper attempts to contribute by focusing on this issue and on empirical techniques that apply to knowledge-based tools.

As the discipline of human-computer interaction grows, the potential availability of techniques increases. Understanding of the use of video protocols [36] and of techniques that apply empirical analysis in the field [29] are seen. The study of software tools can benefit from greater attention to the potential.

6. Conclusion

The goal of empirical study is to serve the co-development of conceptual frameworks and tools. Experience or long-term observations help to identify a conceptual framework which in turn helps to isolate requirements for tools and further observation. Tools provide objects to think and work with, providing concrete tests of conceptual frameworks and methodologies.

While it would be impossible to enumerate all the empirical techniques that are potentially applicable to evaluating knowledge-based software engineering tools, this paper has presented a few common techniques in a format that others can apply. These techniques were illustrated as they were employed in an actual development process.

Acknowledgments

I thank Gerhard Fischer for providing the environment in which this work took place. His adherence to informal evaluation was counterbalanced by Clayton Lewis. Together with Robert Rist and John Rieman, they have shaped my understanding of the role of empirical studies. Thanks also to Frank Shipman and other members of the HCC Group at the University of Colorado for their criticisms. The work was supported in part by Grant No. IRI-0915441 from the National Science Foundation, Grant No. ARI MDA903-86-C0143 from the Army Research Institute, and Grant No. TT-93-006 jointly from US West and the Colorado Advanced Software Institute.

References

[1] R. Alterman, ‘‘Adaptive Planning’’, *Cognitive Science*, Vol. 12, No. 3, 1988, pp. 393-421.

[2] V.R. Basili, ‘‘A Plan for Empirical Studies of Programmers,’’ in *Empirical Studies of Programmers*, E. Soloway, S. Iyengar, eds., Norwood, NJ: Ablex Publishing Corporation, 1986, pp. 252-255, ch. 17.

[3] R. Brooks, ‘‘Towards a theory of the comprehension of computer programs’’, *International Journal of Man-Machine Studies*, Vol. 18, No. 6, June 1983, pp. 543-554.

[4] B. Curtis, ‘‘Five Paradigms in the Psychology of Programming,’’ in *Handbook of Human-Computer Interaction*, M. Helander, ed., Amsterdam: North-Holland, 1991, pp. 87-105, ch. 24.

[5] B. Curtis, E. Soloway, R. Brooks, J. Black, K. Ehrlich, H. Ramsey, ‘‘Software Psychology: The Need for an Interdisciplinary Program’’, *Proceedings of the IEEE*, Vol. 74, No. 8, 1986, pp. 1092-1106, Reprinted in: R.M. Baecker, W.A.S. Buxton (eds): ’Readings in Human-Computer Interaction: A Multidiciplinary Approach’, Morgan Kaufmann Publishers, Los Altos, CA, pp. 150-164, 1987

[6] B. Curtis, H. Krasner, N. Iscoe, ‘‘A Field Study of the Software Design Process for Large Systems’’, *Communications of the ACM*, Vol. 31, No. 11, November 1988, pp. 1268-1287.

[7] D.E. Egan, ‘‘Individual Differences In Human-Computer Interaction,’’ in *Handbook of Human-Computer Interaction*, M. Helander, ed., Amsterdam: North-Holland, 1991, pp. 543-568, ch. 24.

[8] K.A. Ericsson, H.A. Simon, *Protocol Analysis: Verbal Reports as Data*, The MIT Press, Cambridge, MA, 1984.

[9] G. Fischer, S.R. Henninger, D.F. Redmiles, ‘‘Cognitive Tools for Locating and Comprehending Software Objects for Reuse’’, *Thirteenth International Conference on Software Engineering (Austin, TX)*, IEEE Computer Society Press, ACM, IEEE, Los Alamitos, CA, 1991, pp. 318-328.

[10] D.P. Freedman, G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, Dorset House Publishing, New York, 1990.

[11] D. Huff, *How to Lie with Statistics*, Norton, New York, 1958.

[12] ISSCO (Integrated Software Systems Corporation), *The DISSPLA User’s Guide*, San Diego, CA, 1981.

[13] J. Kolodner, *Case-Based Reasoning*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[14] B.P. Korin, *Introduction to Statistical Methods*, Winthrop Publishers, Inc., Cambridge, 1977.

[15] C.H. Lewis, ''Using the 'Thinking-Aloud' Method in Cognitive Interface Design'', Tech. report RC 9265, IBM, 1982.

[16] C. Lewis, ''Some Learnability Results for Analogical Generalization'', Tech. report CU-CS-384-88, Department of Computer Science, University of Colorado, January 1988.

[17] C.H. Lewis, P. Polson, C. Wharton, J. Rieman, ''Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces'', *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*, ACM, New York, April 1990, pp. 235-242.

[18] H.R. Lindman, *Analysis of Variance in Complex Experimental Designs*, W.H. Freeman and Company, San Francisco, 1974.

[19] M. Majidi, D. Redmiles, ''A Knowledge-Based Interface to Promote Software Understanding'', *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, IEEE Computer Society Press, Los Alamitos, CA, September 1991, pp. 178-185.

[20] S. Mannes, W. Kintsch, ''Routine Computing Tasks: Planning as Understanding'', *Cognitive Science*, Vol. 3, No. 15, 1991, pp. 305-342, also published as Technical Report No. 89-8, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[21] R. McCall, ''PHI: A Conceptual Foundation for Design Hypermedia'', *Design Studies*, Vol. 12, No. 1, 1991, pp. 30-41.

[22] J.S. Olson, S.K. Card, T.K. Landauer, G.M. Olson, T. Malone, J. Leggett, ''Computer Supported Cooperative Work: Research Issues for the 90s'', Technical Report 46, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[23] N. Pennington, ''Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs'', *Cognitive Psychology*, Vol. 19, 1987, pp. 295-341.

[24] H. Petroski, *To Engineer Is Human: The Role of Failure in Successful Design*, St. Martin's Press, New York, 1985.

[25] P.L. Pirolli, J.R. Anderson, ''The Role of Learning from Examples in the Acquisition of Recursive Programming Skills'', *Canadian Journal of Psychology*, Vol. 39, No. 2, 1985, pp. 240-272.

[26] D.F. Redmiles, ''Explanation to Support Software Reuse'', *Proceedings of the AAAI 90 Workshop on Explanation (Boston, MA)*, AAAI, Menlo Park, CA, July 1990, pp. 20-24.

[27] D.F. Redmiles, ''From Programming Tasks to Solutions -- Bridging the Gap Through the Explanation of Examples'', Ph.D. Dissertation, Department of Computer Science, University of Colorado, 1992, Also available as TechReport CU-CS-629-92

[28] D.F. Redmiles, ''Reducing the Variability of Programmers' Performance Through Explained Examples'', *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, ACM, 1993, pp. 67-73.

[29] J. Rieman, ''The Diary Study: A Workplace-Oriented Research Tool to Guide Laboratory Efforts'', *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, ACM, 1993, pp. 321-326.

[30] R.C. Schank, *Tell Me a Story: A New Look at Real and Artificial Memory*, Charles Scribner's Sons, New York, 1990.

[31] P.G. Selfridge, L.G. Terveen, M.D. Long, ''Managing Design Knowledge to Provide Assistance to Large-Scale Software Development'', *Proceedings of the 7th Annual Knowledge-Based Software Engineering (KBSE-92) Conference (McLean, VA)*, IEEE Computer Society Press, Los Alamitos, CA, September 1992, pp. 163-170.

[32] B.A. Sheil, ''The Psychological Study of Programming'', *Computing Surveys*, Vol. 13, No. 1, March 1981, pp. 101-120.

[33] B.A. Sheil, ''Power Tools for Programmers'', *Datamation*, February 1983, pp. 131-143.

[34] E. Soloway, K. Ehrlich, ''Empirical Studies of Programming Knowledge'', *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984.

[35] P. Sprent, *Applied Nonparametric Statistical Methods*, Chapman and Hall, London, 1989.

[36] L.A. Suchman, R.H. Trigg, ''Understanding Practice: Video as a Medium for Reflection and Design,'' in *Design at Work: Cooperative Design of Computer Systems*, J. Greenbaum, M. Kyng, eds., Hillsdale, NJ: Lawrence Erlbaum Associates, 1991, pp. 65-89, ch. 4.

[37] D.B. Walz, J.J. Elam, H. Krasner, B. Curtis, ''A Methodology for Studying Software Design Teams: An Investigation of Conflict Behaviors in the Requirements Definition Phase,'' in *Empirical Studies of Programmers: Second Workshop*, G.M. Olson, E. Soloway, S. Sheppard, eds., Norwood, NJ: Ablex Publishing Corporation, Lawrence Erlbaum Associates, 1987, pp. 248-263.

[38] M. Weiser, ''Some Computer Science Issues in Ubiquitous Computing'', *Communications of the ACM*, Vol. 36, No. 7, July 1993, pp. 74-84.