Reducing the Variability of Programmers' Performance Through Explained Examples

David F. Redmiles

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder, Colorado 80309
redmiles@cs.colorado.edu

ABSTRACT
A software tool called EXPLAINER has been developed for helping programmers perform new tasks by exploring previously worked-out examples. EXPLAINER is based on cognitive principles of learning from examples and problem solving by analogy. The interface is based on the principle of making examples accessible through multiple presentation views and multiple representation perspectives. Empirical evaluation has shown that programmers using EXPLAINER exhibit less variability in their performance compared to programmers using a commercially available, searchable on-line manual. These results are related to other studies of programmers and to current methodologies in software engineering.

KEYWORDS: software engineering, user interface, knowledge representation, semantic networks, learning, analogy, programming plans.

INTRODUCTION: EXAMPLES IN PROGRAMMING

Programming requires a person to transform a general, informal understanding of a goal into a formal model using operators and components interpretable by the computer [4, 10, 13]. Program examples can aid this task by illustrating results of applying and combining specific components. When the results or goals of an example match a programmers' current goals for a task, the example provides a means of reducing the space of possible components the programmers might need, as well as illustrating how to use them. Thus, the problem-solving strategy being advocated here for programming is to use examples by analogy.

Empirical evidence of the viability of an example-based approach has come out of cognitive studies of learning and problem solving. Lewis observed that people learning procedures can use and generalize from even a single example, relying on heuristics and some background knowledge [11]. Pirolli and Anderson observed that examples played a crucial role for students learning LISP concepts [14]. In another study, Kessler and Anderson noted variability in learners' performance and concluded that when relying on examples, learners must be able to construct correct mental models of examples [9]. Similarly, Chi and colleagues observed that students can learn well from examples, provided they are careful to explain the examples to themselves as they learn (i.e., develop good mental models) [2].

Together, these different studies imply a uniform result: learners who develop good mental models of examples can apply or transfer that knowledge to new tasks. Thus, to assist programmers, it is not always sufficient to simply provide examples; some assistance aiding the development of a mental model of the example may also be required.

This conclusion is the fundamental assumption of a programming tool called EXPLAINER. EXPLAINER combines code examples with knowledge equivalent to the programming plan [20, 19, 17] underlying the example. Through a hypermedia interface, programmers can explore the relationship between the abstract programming plan and the concrete implementation components in an example related to their programming task. An empirical evaluation has shown that providing the knowledge behind an example, namely the programming plan, greatly reduced the variability of programmers' performance.

The remaining sections of this paper provide the background for the EXPLAINER research, illustrate how the EXPLAINER interface is used, describe how knowledge is represented and input, and report specific results of the empirical evaluation. Some of the sections are necessarily brief. In-depth information is found in [15].

AN EXAMPLE-BASED DESIGN PROCESS

The EXPLAINER tool is only one component of an example-based design process (see Figure 1). Examples residing in a catalog repository must be retrieved by users. Retrieval requires users to articulate some specification of requirements for the current task (such as the task in Figure 2) in order for an appropriate example to be delivered (such as the example in Figure 3). Once an example is retrieved, users must be supported in potential modification of the example to adapt it to their new task. Working with an example can lead to additional ideas about the current task, possibly leading to refinement of the original requirements specification and retrieval of additional examples.

This model of design, the interrelationship of the different components, and current systems implementations are discussed in detail elsewhere [6]. It is presented here for context and to emphasize that the examples presented to users by EXPLAINER are selected for their applicability to a programmer or designer's current task. Knowing that an example is related to their task helps people develop the analogy from example to task, which is critical to the example-based approach [7].

The features or concepts of an example that are relevant to different tasks can vary. However, the use of analogy has been simplified in our exploratory work by restricting the domain of the tasks and examples to graphics functions. Thus, in a sense, the existence of a common perspective between example and task is guaranteed; minimally, it is found in the graphics features of an example (i.e., circles, labels, etc.). Once programmers recognize the features they want in a retrieved example, they can use the EXPLAINER interface to explore how features are mapped onto programming concepts. This decomposition is essentially the programming plan for the example [20, 17] (See also [16]).

EXPLAINER USER INTERFACE

The purpose of the EXPLAINER interface (see Figures 3 and 4) is to make apparent to the programmer the relationship between abstract programming plans and the specific implementation components [13, 19]. The interface was implemented as a hypermedia tool. This implementation allows minimal information about the example to be initially presented [5, 1]. The programmers can then decide which specific features of the example they want to explore, presumedly choosing those most relevant to their current task. Information is accessed and expanded through a command menu (see Figure 4a). Almost all items presented on the screen are mouse sensitive and are equally accessible for command actions.

The interface presents multiple presentation *views* of the information comprising an example, code listing, sample execution, component diagrams, and text. These views are initially selected for EXPLAINER due to their popularity in existing computer-aided software engineering (CASE) tools [18]. Unique to EXPLAINER is the characteristic that the same information is presentable in different views. For

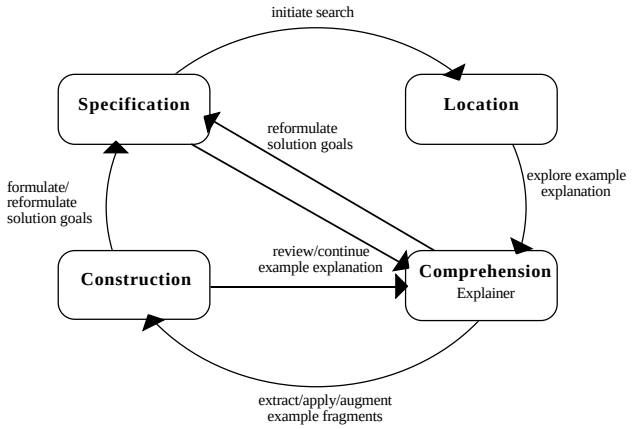


Figure 1: An Example-Based Design Process

Clock Programming Task
Write a program to draw a clock face (without the hands) that shows the hour numerals, 1 - 12. Your solution should look like the picture below.

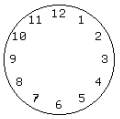


Figure 2: Clock Task as Described to Programmers

instance, in the Cyclic Group Example shown in Figure 3, the LISP concept of the call to the function that draws the circle is presented as a stylized fragment of the code listing, as a circle graphic object in the sample execution, or as node in a component diagram. The possibility to view the example information in different external forms accommodates different individual preferences. Redundant views also provide reinforcement of new concepts.

Within each view, the programmer can access information from different representation *perspectives.* Currently, the different perspectives are selected from a list (Figure 4b), appearing after a command is selected (Figure 4a). For instance, in the diagram view, the programmer has created diagrams of the example from the Plot Features and LISP perspectives (upper right of Figure 3—the LISP perspective being only partially visible). Text has been presented from LISP, Program Features, and Cyclic Operations perspectives (lower right of Figure 3).

Thus, the EXPLAINER interface allows programmers to access information about programming plans through different views and from different perspectives. Highlighting and textual descriptions allow programmers to understand the relationships between elements of programming plans and system components. Figure 3 is the actual state of the EXPLAINER interface as produced by one of the test subjects during the evaluation. The subject was exploring this example program to perform the programming task of

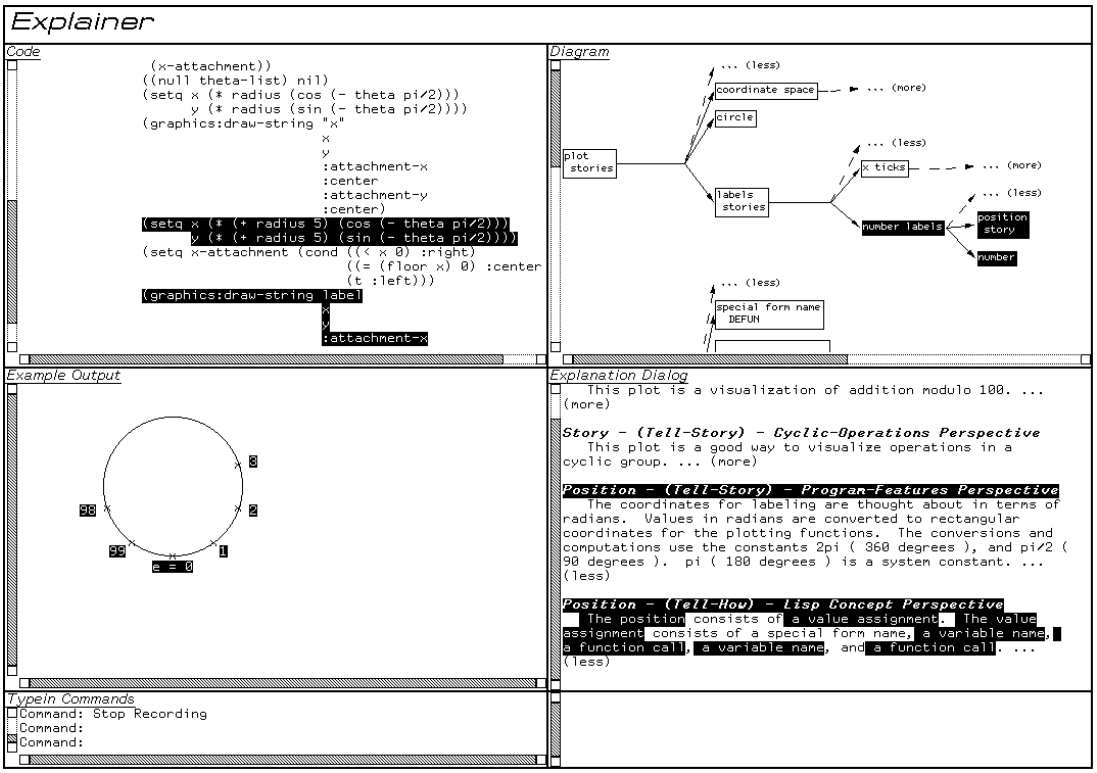


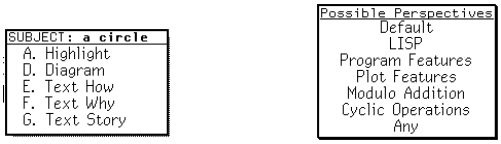
Figure 3: Exploring the ‘‘Cyclic Group’’ Example in EXPLAINER

The above screen shows the actual state of the EXPLAINER interface at the end of one programmer’s test session. The programmer was exploring the information in this example in learning how to complete the programming task of Figure 2.

Figure 2. The screen shows that the subject had accomplished the following:

- expanded the diagram view to explore the components of labels in the Plot Features perspective (clicking on ‘‘(more)’’ cues);
- redrawn the initial diagram from its Plot Features perspective to a LISP perspective—only a portion is visible in the middle of the diagram pane (menu action ‘‘Diagram’’);
- retrieved the story, displayed in the explanation-dialog view, about the concept of labels in the Program Features perspectives (menu action ‘‘Text Story’’);
- generated a description of how labels are implemented in the LISP perspective (menu action ‘‘How’’); and
- highlighted the concepts having to do with labels common across the several different perspectives and the four views.

This specific information enabled the test programmer to identify the LISP function called to draw the label, the assignment function that calculated the position, and what variables the position calculation depended on. The programmer could then apply the same functions in the solution of the clock task, or in this case, simply modify a copy of the example to place the labels inside the perimeter of the circle.



(a) **(b)**

Figure 4: Pop-up Menus in EXPLAINER

KNOWLEDGE REPRESENTATION, USE, AND INPUT

The representation of plan information in EXPLAINER is accomplished using semantic networks (see Figure 5). In EXPLAINER, nodes in a semantic network correspond to *concepts* in a given *perspective.* A perspective is a theme or point of view under which the example may be described. Three perspectives appear in the scenario: a programming language perspective of LISP concepts, a Plot Features perspective, and a problem-domain perspective of Cyclic Operations in group theory. Concepts in the LISP perspective consist of function calls and arguments. Concepts in the Plot Features perspective consist of graphic concepts such as ‘‘number labels’’ and ‘‘x ticks.’’ Concepts in the Cyclic Operations perspective consist of problem-domain concepts such as ‘‘operator’’ and ‘‘elements.’’

The networks connect nodes through three kinds of links: *components, roles,* and *perspectives.* The components link connects one concept to zero or more concepts that may

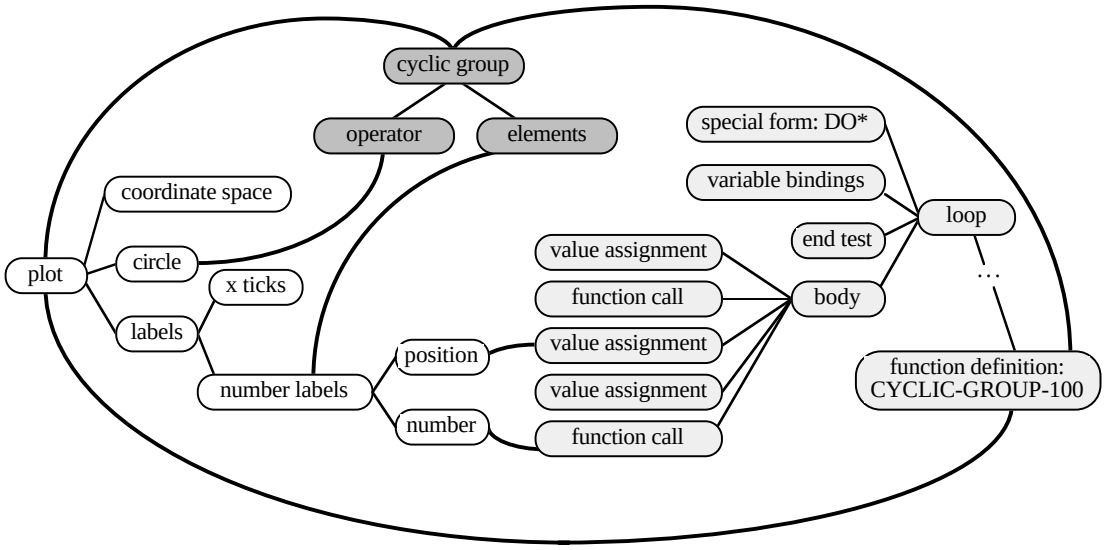


Figure 5: Partial Repesentation of an Example

Three perspectives are shown in this figure: Plot Features (unshaded ovals), LISP concept (shaded ovals), and Cyclic Operations (darkest ovals). The straight lines represent components/roles links. The arcs correspond to perspectives links.

comprise it. In the above example, the ''plot'' consists of a ''coordinate space,'' a ''circle,'' and ''labels.'' The components link captures the ''how to'' or implementation knowledge in a example. The roles link is the inverse of the components link and supples one kind of ''why'' or goal knowledge. For example, a ''circle'' is drawn as part of the ''plot.'' Concepts are identified with one specific perspective. However, they can have equivalent or analogous counterparts in other perspectives. This relationship is captured in the perspectives link and provides the information used for highlighting the correspondences between the different perspectives. For example, the ''position'' and ''number'' components of the ''number labels'' concept in the Plot Features perspective are equivalent to a ''value assignment'' and a ''function call'' respectively in the LISP perspective.

In sum, concepts in different perspectives are composed into semantic networks. The networks in different perspectives are interrelated through the perspectives links of individual concepts. Thus, as a whole, an example program is one semantic network that may be interpreted according to various perspectives.

Essentially all of the implementation of EXPLAINER is devoted to supporting the user interface. Since all of the commands are accessing information about an example, they are always accessing concepts in the semantic network representing that example. The minimal information presented initially on the screen provides starting points of concepts in this network. The actions on the command menu initiate searches through the network and expose additional information. Applying the ''Text How'' command to a concept causes that concept's components to be formated (through simple patterns) and presented in a sentence. Using the ''Highlight'' command exposes all the perspective links from a specified concept by highlighting equivalent concepts in all the views. The search pattern is basically breadth-first, though various characteristics of the search path can be controlled to find concepts of specific perspectives, views, or distances.

The scheme for construction of new examples relies on work done with Andreas Girgensohn in conjunction with his research on systems for end-user modifiability [8]. Additional examples are found in [6]. Briefly, the input of knowledge about examples is a semiautomated process. The creator of the knowledge is currently assumed to be the original author of the example. Once an example program has been written, its code is parsed automatically into a semantic net of LISP concepts. Higher-level perspectives can be created by a concept input menu (not shown). Using a concept editing menu (see Figure 6), concepts from different perspectives can be equated. For instance, as shown in the figure, the concept ''elements'' of the Cyclic Operations perspective (highlighted in the diagram pane) can be equated to the LISP perspective concept ''label-list'' (highlighted in the code pane). The network of concepts associated with a program example is saved together with the example.

With all knowledge-based interfaces, the issue of the cost of entering knowledge arises. The techniques discussed in the preceding paragraph simplify this process. The information for the Cyclic Group Example required 2.5 hours to input after about 5 hours of planning the content of the example. The example introduces 30 new concept classes while reusing 63 existing classes. As the number of examples collected increases, the number of new classes needed is expected to decrease. The Cyclic Group Ex-

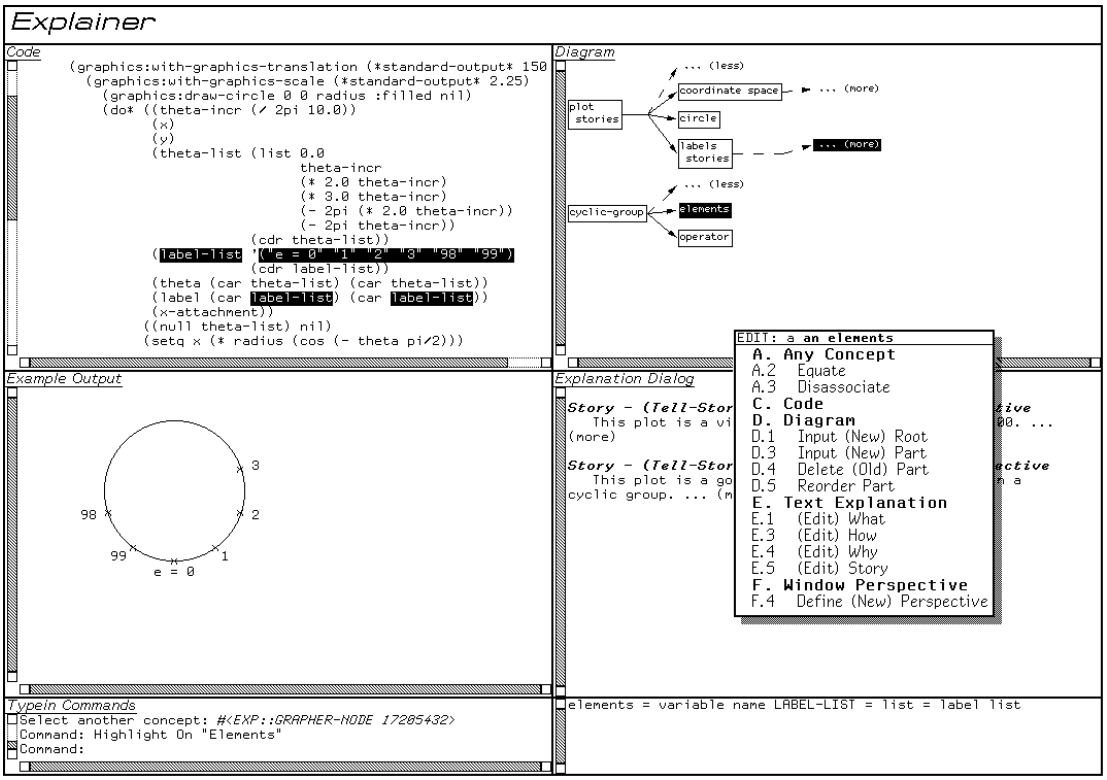


Figure 6: Authoring an Example in EXPLAINER

Two parts, ‘‘elements’’ and ‘‘operator’’ are defined and added to the new concept ‘‘cyclic group’’ through the concept editing menu. The ‘‘elements’’ part is then equated to the code concept of ‘‘label-list’’ through the same menu.

ample illustrates 6 of 70 components in a graphics library. While a systematic analysis of ‘‘coverage’’ has not be performed, the total number of examples for adequately illustrating this library is expected to be small.

EMPIRICAL EVALUATION

The empirical evaluation of EXPLAINER tested three conditions under which subjects solved the programming task of drawing a clock face (see Figure 2). In all three conditions, programmers were given the same example, a program illustrating operations in a cyclic group (see Figure 3). The conditions varied the programming tool that the three groups of subjects worked with to help them understand the example. In the first condition, subjects worked with the EXPLAINER tool as described above. In the second condition, subjects also worked with the EXPLAINER tool; however, the interactive menu was deactivated (the example information was fully expanded when subjects began the test). In the third condition, subjects worked with a commercially available, searchable on-line manual which contained descriptions of all the functions used in the example. The purpose of the intermediary second condition was to determine if only the difference in information content between EXPLAINER and the on-line manual affected the results. In Tables 1-4, the three conditions are identified as ‘‘E’’, ‘‘O’’, and ‘‘D’’ for EXPLAINER, menu-Off EXPLAINER, and commercial documentation.

The evaluation measured the performance of programmers with respect to variability in two senses. First, a notion of ‘‘directness’’ was defined as the number of different variations programmers would try in solving a task. The rationale was that the more support the programmers had from the example and tool in solving the task, the lower the number of trial and error variations would be. The observed measures are defined below and these are compared across groups. Second, within each group the variance in the observed measures is calculated and then compared across groups. The rationale with this test was that the more support the example and tool combination provided, the more uniform (smaller variance) the programmers’ behavior as measured would be within a condition.

Eight subjects were tested for each condition and were randomly assigned to conditions. The subjects all had roughly the same background knowledge in LISP programming, being recruited from masters-level artificial intelligence classes in computer science. However, they were not already familiar with the graphics functions required by the task. Despite the equivalent backgrounds, one subject in the on-line documentation group (condition D) dropped out after working for 26 minutes. That subject’s data are not reported in the tables because of the requirements of the calculations; if included, they would shift the results further in EXPLAINER’s favor. Thus the frequencies represented in the tables are 8, 8, and 7 for conditions E, O, and D.

Table 1: Means of Session Data by Condition

	E	**O**	**D**
Points Completed	4.63	4.63	4.71
Solution Time	43.25	39.00	36.14
Changes	7.38	9.75	13.14
Runs	4.75	6.50	9.57

Table 2: Means of Ratio Data by Condition

	E	**O**	**D**
Changes/Points	1.57	2.06	3.14
Runs/Points	1.03	1.41	2.45
Solution Time/Points	9.41	8.96	8.64

Table 3: Variances of Session Data Compared by Squared-Rank Test (Explainer Against Document-Examiner)

	E	**D**	**|Z|**	$p <$
Points Completed	0.55	0.57	1.92	.1
Solution Time	175.93	445.14	1.17	—
Changes	4.55	67.14	2.91	.01
Runs	2.21	78.29	2.91	.01

Table 4: Variances of Ratio Data Compared by Squared-Rank Test (Explainer Against Document-Examiner)

	E	**D**	**|Z|**	$p <$
Changes/Points	0.08	7.86	2.72	.01
Runs/Points	0.08	9.71	2.70	.01
Solution Time/Points	6.41	56.61	1.70	.1

The means of the performance measures are summarized in Table 1. Points Completed measures how many of five subtasks were completed by subjects. The subtasks were identified by steps that would be required to solve the Clock Task. Identifying the subtasks allowed partial grading of tasks not fully completed and allowed a more realistic programming situation in which subjects would not necessarily know a priori what subtasks would have to be performed. Partial solutions are compensated for by ratios described below. Solution Time measures how long a subject worked before declaring their solution complete. Changes measures how many attempts a subject made before completing the task. Runs measures how many trial executions were made (i.e., to verify changes).

To compensate for the fact that subjects might complete different numbers of subtasks, ratios of the different measures per Points Completed were calculated for each subject. The means of the resulting ratios are summarized in Table 2.

The first evaluation of variability considered the measurements of Tables 1 and 2. On first inspection, the values of the two tables present a mixed picture. Table 1 indicates that none of the groups differed greatly in the degree of their solutions (Points Completed) nor in the time taken (Solution Time). However, the number of variations made and tried (Changes and Runs) tended to increase from condition E to O to D. This trend is borne out in the ratios in Table 2. The EXPLAINER group appeared to complete the programming task more ‘‘directly’’ while the documentation group proceeded according to a ‘‘trial-and-error’’ method. Furthermore, the Points Completed measure is actually inflated for condition D since the eliminated subject did not complete any goals. Including that observation reduces the mean for Condition D for Points Completed to 4.13. Unfortunately, the standard ANOVA test did not indicate a significant difference in means for the measures, including Changes and Runs.

The second evaluation of variability considered the variances of the measures within each condition. Tables 3 and 4 show variances for the different measures and the significance yielded by the Squared-Rank Test. Except for Solution Time, the difference in variances was statistically significant. For brevity, only the comparison between the two extreme conditions (E and D) is shown here. In general, the trend of increased variance from condition E to O to D holds with statistical significance. Detailed values and comparisons are available in [15].

As a group, then, the subjects using EXPLAINER performed the programming task more directly, and they performed it with less inter-subject variability. Subjects using the online documentation tool proceeded in a trial-and-error fashion and, not surprisingly, exhibited great inter-subject variability. It is important to note that the reduction in variance was not at the cost of performance. The ‘‘better’’ end-point values (high values for Points Completed, low for other measures) of the ranges of the measures were similar—good performers were about the same in all conditions. The reduction in variance resulted from ‘‘worse’’ end-point values coming closer to the ‘‘better’’ end—otherwise poor performers were helped by EXPLAINER.

CONCLUSIONS

The variability of the programmers’ performance in the online documentation group is consistent with other studies of programmers (see the survey by Egan [3]). Furthermore,

the provision of an example by itself was insufficient to prevent this variability, as also observed by Kessler and Anderson and noted earlier. However, the provision of an example, supplemented by information constituting a representation of a programming plan and with a means of exploring the relationship of the programming plan to a specific example solution, did stem the variability. Programmers who needed to compensate for variation in background knowledge, skill, or other predispositions, were supported by the EXPLAINER tool and approach.

The kind of support that the EXPLAINER tool provides is becoming increasingly important in the software field. Consider the widely-acclaimed methodology of software reuse [12]. The premise is that ever-growing repositories (subroutine libraries or software packages) of proven components (subroutines, functions, object classes, etc.) are made available to programmers to use as operators in new programs. Though this approach does save a programmer the effort of redevelopment, the essential cognitive problem of programming remains: programmers still must know when and how to apply which components or operators to achieve specific results [4]. This knowledge is central to the examples as represented by EXPLAINER.

ACKNOWLEDGMENTS

I thank all of the students who participated in the empirical study. I thank Gerhard Fischer, Clayton Lewis, John Rieman, and Robert Rist for their comments and support. This research was funded in part through grants from the Army Research Institute (ARI MDA903-86-C0143) the National Science Foundation (IRI-9015441), and the Colorado Advanced Software Institute (TT-93-006).

REFERENCES

1. J.B. Black, J.M. Carroll, S.M. McGuigan. What Kind of Minimal Instruction Manual Is The Most Effective. *Human Factors in Computing Systems and Graphics Interface, CHI+GI'87 Conference Proceedings (Toronto, Canada)*, ACM, New York, 1987, pp. 159-162.

2. M.T.H. Chi, M. Bassok, M. Lewis, P. Reimann, R. Glaser. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science 13*, 2 (1989), 145-182.

3. D.E. Egan. Individual Differences In Human-Computer Interaction. In M. Helander (Ed.), *Handbook of Human-Computer Interaction*, North-Holland, Amsterdam, 1991, Chap. 24, pp. 543-568.

4. G. Fischer. Cognitive View of Reuse and Redesign. *IEEE Software, Special Issue on Reusability 4*, 4 (July 1987), 60-72.

5. G. Fischer, T. Mastaglio, B.N. Reeves, J. Rieman. Minimalist Explanations in Knowledge-Based Systems. Jay F. Nunamaker, Jr (Ed.), *Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track*, IEEE Computer Society, 1990, pp. 309-317.

6. G. Fischer, A. Girgensohn, K. Nakakoji, D. Redmiles. Supporting Software Designers with Integrated, Domain-Oriented Design Environments. *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Engineering 18*, 6 (1992), 511-522.

7. M.L. Gick, K.J. Holyoak. Analogical Problem Solving. *Cognitive Psychology 12* (1980), 306-355.

8. A. Girgensohn. *End-User Modifiability in Knowledge-Based Design Environments*. Department of Computer Science, University of Colorado, Boulder, CO, 1992. Also available as TechReport CU-CS-595-92.

9. C.M. Kessler, J.R. Anderson. Learning Flow of Control: Recursive and Iterative Procedures. *Human-Computer Interaction 2* (1986), 135-166.

10. W. Kintsch, J.G. Greeno. Understanding and Solving Word Arithmetic Problems. *Psychological Review 92* (1985), 109-129.

11. C. Lewis. Why and How to Learn Why: Analysis-Based Generalization of Procedures. *Cognitive Science 12*, 2 (1988), 211-256.

12. B. Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software 4*, 2 (March 1987), 50-64.

13. N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology 19* (1987), 295-341.

14. P.L. Pirolli, J.R. Anderson. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology 39*, 2 (1985), 240-272.

15. D.F. Redmiles. *From Programming Tasks to Solutions -- Bridging the Gap Through the Explanation of Examples*. Department of Computer Science, University of Colorado, Boulder, CO, 1992. Also available as TechReport CU-CS-629-92.

16. C.H. Rich, R.C. Waters. *The Programmer's Apprentice*. Addison-Wesley Publishing Company, Reading, MA, 1990.

17. R.S. Rist. Schema Creation in Programming. *Cognitive Science 13* (1989), 389-414.

18. J. Sodhi. *Software Engineering Methods, Management, and CASE Tools*. TAB Professional and Reference Books, Blue Ridge Summit, PA, 1991.

19. E. Soloway, J. Pinto, S. Letovsky, D. Littman, R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM 31*, 11 (November 1988), 1259-1267.

20. E. Soloway, K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 595-609.