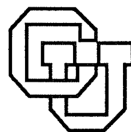
From Programming Tasks to Solutions - Bridging the Gap through the Explanation of Examples

David F. Redmiles

CU-CS-629-92 July 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Abstract

Evidence, experience, and observation indicate that examples provide a powerful aid for problem solvers. In the domain of software engineering, examples not only provide objects to be reused but also a context in which programmers can explore issues related to their current task. This dissertation describes a software tool called EXPLAINER, which supports programmers' use of examples in the domain of graphics programming, assisting them with examples and explanations from various views and representation perspectives. EXPLAINER provides a conceptual and working framework for the study of programmers' uses of examples in problem solving and serves as a test bed for representations based upon multiple perspectives. The EXPLAINER approach is evaluated and compared with other available approaches, such as on-line manuals. The evaluation showed that subjects using EXPLAINER exhibited a more controlled and directed problem-solving process compared to subjects using a commercially available, searchable on-line manual. Representation of examples from multiple perspectives is seen as a critical aspect of catalog-based design environments.

Acknowledgements

This dissertation is a culmination of many years of study and work as a professional and there are many people I would like to acknowledge and thank. Foremost, I thank John Rieman for his constant interest and critiquing of my work, as well as being a friend over the past three years. His selfless effort in reviewing this dissertation in particular has greatly improved its quality. I would like to thank Christian Rathke for his great effort two years ago, helping me refine my dissertation proposal, and for his continuing interest and support since the start of my Ph.D. studies, five years ago. I would like also to thank Robert Rist for many helpful ''discussions'' about the proper approach to experiments during his recent sabbatical at C.U. I thank Mehran Majidi who spent many hours carrying out some of the informal observations of students using EXPLAINER.

I would like to thank all the members of my dissertation committee, Gerhard Fischer, my advisor, Clayton Lewis, Jim Martin, Walter Kintsch, and Ray McCall. Clayton Lewis helped me and influenced my thinking especially through his uncanny ability to always ask the right questions. I thank Jim Martin that his door was always open. I would like to thank my friends and family for their support: Andreas Girgensohn, Frank Shipman, Kumiyo Nakakoji, Christopher Runyan, Christina Bcyzkowski, Joanne Murray, Jim Sims, Greg Rhoads, Richard Apperson, and Joseph, Melvin, and Margaret Redmiles. Thanks to Gerry Schnackenberg, Anne Schwarz, and many others for their intercessary prayers.

To come full circle, I thank again Gerhard Fischer. Without the environment and opportunity he provided, these ideas could not have flourished. This research was supported in part through grants from the National Science Foundation (NSF Grant IRI-9015441), the Army Research Institute (ARI Grant ARI MDA903-86-C0143), and US West.

Table of Contents

1. Introduction—Programming with the Help of Examples **1**
1.1 Two Scenarios for Programming Support 1
1.2 Examples in Programming 2
1.3 An Example-Based Approach to Design 3
1.4 The Explainer System 4
1.5 Evaluation 5
1.6 Reader's Guide 6

2. Conceptual Framework for Example-Based Design **7**
2.1 Overview 7
2.2 Problems in Software Engineering 7
2.3 Examples to Support Programming 8
2.4 Representing and Explaining Examples 9
2.5 Summary 10

3. Approach of the EXPLAINER System **11**
3.1 An Example-Based Model of Design 11
3.2 Specification and Location 12
3.3 Comprehension 14
3.3.1 Perspectives, Views, and Explanations 14
3.3.2 Judging Relevance 15
3.3.3 Building Analogy 16
3.4 Construction 18
3.5 Summary 18

4. Implementation—Representation and Interpretation **21**
4.1 Overview 21
4.2 System Goals 21
4.3 Users' Scenario 21
4.4 Implementation Architecture 24
4.5 Representation of Examples 26
4.6 Interpretation—Parameters for Search 27
4.7 Summary 28

5. Initial Observations **29**
5.1 Informal Observations 29
5.2 Observations with a Human Consultant 29
5.3 Observations with the First System 32
5.4 Summary 35

6. Evaluation through a Formal Experiment **36**
6.1 Goals 36
6.2 Design 36
6.3 Subjects 37
6.4 Materials and Procedure 37
6.5 Measurable Variables 38
6.5.1 Defining the Task and Example 38
6.5.2 Observing Subjects' Programming and Usage of Tools 39
6.5.3 Assessing Subjective Information 40
6.5.4 Defining "Better Performance" 40
6.5.5 Glossary of Variables 41
6.6 Results 42
6.6.1 Subjects' Solution Processes and Tool Usage 42

6.6.2 Overall Quantitative Data 44
6.6.3 Interview Data 47
6.7 Discussion 48
6.7.1 Subjects' Solution Processes and Tool Usage 48
6.7.2 Overall Quantitative Data 49
6.7.3 Interview Data 50
6.8 Summary 51
7. Implications of Observations and Evaluation **53**
7.1 Trends and the Larger Picture 53
7.2 Prototyping Behavior 53
7.3 Effectiveness of Tools 55
7.4 Effort and Payoff in Problem Solving 59
7.5 Summary 60
8. Other Related Work **62**
8.1 Knowledge-Based Software Engineering 62
8.2 Domain Knowledge and Design Recovery 62
8.3 Case-Based Reasoning 62
8.4 Summary 63
9. Conclusion **64**
References **66**
Appendix I. Excerpts of Code **74**
Appendix II. Detailed Information for the Formal Experiment **77**

List of Figures

Figure 1-1: The Clock Task 2
Figure 1-2: System Documentation for the DRAW-CIRCLE Function 2
Figure 1-3: Candidate Documentation Topics for Keyword "labels" 3
Figure 1-4: Explainer's Presentation of the Cyclic Group Example 4
Figure 1-5: An Example-Based Design Process 5
Figure 3-1: Processes and Supporting System Components in Example-Based Design 12
Figure 3-2: Specification Component of CATALOGEXPLORER (reproduced by permission from [Fischer et al. 92a]) 12
Figure 3-3: CODEFINDER User Interface (reproduced by permission from [Fischer, Henninger, Redmiles 91]) 13
Figure 3-4: Explainer's Presentation of the Cyclic Group Example (Repeated from Figure 1-4) 15
Figure 3-5: Final EXPLAINER Screen in a Test Session (Subject #1) 17
Figure 3-6: Adding a New Concept 18
Figure 3-7: Adding Subcomponents to a Concept 19
Figure 3-8: Constructing New Objects in EXPLAINER 20
Figure 4-1: EXPLAINER: Initial Screen 22
Figure 4-2: Pop-up Menus in EXPLAINER 22
Figure 4-3: EXPLAINER: Expanded Objects 23
Figure 4-4: EXPLAINER: Final Screen (Repeated from Figure 3-5) 24
Figure 4-5: EXPLAINER Architecture: Theory, Methods, Knowledge, and Data Flow 25
Figure 4-6: Partial Repesentation of an Example 26
Figure 4-7: Schematic of Concept Representation 27
Figure 5-1: Example and Final Outputs for Clock Task 30
Figure 5-2: Dialog Excerpts from Observations with Human Consultant 31
Figure 5-3: The EXPLAINER Prototype as used in the Student Observations (reproduced by permission from [Majidi, Redmiles 91]) 32
Figure 5-4: Trends in the TMYCIN Observations (reproduced by permission from [Majidi, Redmiles 91]) 34
Figure 6-1: Description of the Clock Task as Given to Programmers 39
Figure 6-2: Actions in EXPLAINER Juxtaposed with Solution Progress 42
Figure 6-3: Actions in Menu-Off EXPLAINER Juxtaposed with Solution Progress 43
Figure 6-4: Actions in DOCUMENTEXAMINER Juxtaposed with Solution Progress 43
Figure 7-1: Complete Alterations vs. Runs—Effects of Tools on Prototyping 54
Figure 7-2: TMYCIN Observations—Effectiveness of Tool 56
Figure 7-3: Formal Experiment—Effectiveness of Tool 57
Figure 7-4: Work Accomplished vs. Solution Time 58
Figure 7-5: Solutions vs. Time According to Groups 59
Figure 7-6: Start-up Time and Projected Correct Alterations 61
Figure II-1: Annotated Listing of the Cyclic Group Example 78
Figure II-2: Annotated Solution of the Clock Task 79
Figure II-3: Post Experiment Questionnaire followed by Interviewer 84
Figure II-4: Post Experiment Questionnaire followed by Interviewer 85

List of Tables

Table 5-1: Summary of Measurements Taken During the TMYCIN Observations 33
Table 6-1: Control Groups in the Clock Experiment 37
Table 6-2: Means of Primary Measures 44
Table 6-3: Means Revised to Factor Out Point F (See Section 6.6.2) 44
Table 6-4: Means of Calculated Ratios 45
Table 6-5: Means of Changes for Alteration Point B 45
Table 6-6: Primary Measures: Medians, Ranges, and 95-Percent Confidence Intervals 46
Table 6-7: Calculated Ratios: Medians, Ranges, and 95-Percent Confidence Intervals 46
Table 6-8: Variances of Primary Measures and Calculated Ratios Compared by the Squared-Rank Variance Test 47
Table 6-9: Means of Measures from Interview 47
Table 6-10: Two-way Comparison of General Ratings from Interview using Kruskal-Wallis Test 48
Table 6-11: Two-way Comparison of Ratings and Ranks between Full and Menu-Off EXPLAINER using Kruskal-Wallis Test 49
Table II-1: Summary of Basic Data 80
Table II-2: Summary and Comparison of Revised Data 81
Table II-3: Summary of Ratio Data 82
Table II-4: Summary of Changes by Individual Alteration Points 83

1. Introduction—Programming with the Help of Examples

1.1 Two Scenarios for Programming Support

How can programmers be helped in their programming tasks? Solutions ranging from documentation to analysis utilities to automatic programming techniques have been offered and continue to be refined. However, programming continues to be a difficult job, and reliance on human experts, either professional consultants or knowledgeable colleagues, continues to be an integral part of software development.

Consider the simple graphics programming task of drawing a clock face, as shown in Figure 1-1. With the facilities of a modern software environment, what kind of support does a programmer have for implementing a program for this task? One approach the programmer might take is to use the system documentation to find what functions might draw circles and labels. Figure 1-2 shows the Symbolics on-line documentation tool, the DOCUMENTEXAMINER. Using it to search for topics related to the keyword ''circles'' yields the list in the upper right of the screen. An experienced programmer would be able to use syntactic clues to eliminate many of the possible candidates and would even recognize that all of the suggested topics for ''labels'' (see Figure 1-3) have nothing to do with the graphics concept of annotating a plot.

After the correct documentation sections have been identified, the question remains: how much help can they be to the programmer? The documentation sections provide information about the calling sequence of the functions in question, list all the possible arguments, and then explain some of the arguments, sometimes providing examples to illustrate variations. However, this information does not necessarily address the situation or problem the programmer faces. Even in a general sense, how to combine functions is not necessarily addressed; the examples illustrate some combinations but do not explain the relationships or effects (e.g., with-room-for-graphics is a commonly used initialization function). The programmer must cross-reference to other sections of the documentation. In so doing, the programmer encounters information further and further removed from the context of the problem.

As this scenario illustrates, when the programmer relies primarily on documentation for programming support, the organization and style of the documentation is critical. The Symbolics documentation is organized according to a common explanation style that may be termed *principles first*. The disadvantages of this style are the large number of possible topics, the quantity of information in descriptions, the lack of relevance within topic descriptions, the dispersion of relevant information across topics, and the failure to address an overall plan for the task.

A second approach the programmer may take is to simply ask a colleague who might know how to solve the task. The colleague might be able to provide a plan or specification, perhaps even at the detailed level of which functions to call. He or she might at least be able to narrow down which sections of the systems documentation to read. Perhaps they have solved a similar problem before and can provide and explain that code. There may still be work to be done, learning some new information, understanding and modifying existing code; but much progress has been made compared to the first approach. The colleague has reduced the number of topics to be examined, pointed out relevant areas, cross-referenced information, and even provided in part or whole an overall plan through specification or code example.

In both scenarios, there is little difference in the materials available for supporting the solution, but there is a significant difference in how the materials are accessed and presented. The assisting colleague has smoothed the process by his or her understanding of the programming task. That expertise makes the colleague one of the most valuable programming support aides available. Brooks downplays the value of many programming tools compared to the human resource [Brooks 87]. In his terminology, the materials underlying both scenarios above would be supporting the *accidents* of the software engineering process, the human would be affecting the *design*.

The research presented here describes a software tool EXPLAINER (Figure 1-4), but the focus of the approach is on the human factors and on the design process. In particular, the tool tries to capture aspects of a human expert assisting a programmer by providing explanations of existing examples relevant to the programmer's current task—the second alternative described above. The research observes the reactions of programmers to the tool as an assistant and compares performance using the tool to performance with

other support, the DOCUMENTEXAMINER in particular. The knowledge-representation needs of the tool are discussed and its implementation is described. In sum, how the tool fits into the design process, how the tool provides some of the knowledge human assistants provide, and how programmers react to the tool as an assistant are the issues this research addresses.

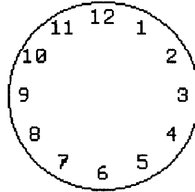


Figure 1-1: The Clock Task

The "clock" task is a simple programming task used to illustrate ideas throughout this thesis. It was also used during testing with programmers. This task is described to programmers roughly as follows: 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 above.

Document Examiner

graphics:draw-circle *center-x center-y radius* &key (*inner-radius* **0**) (*start-angle* **0**) (*end-angle* **graphics:2pi**) (*clockwise* **nil**) (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*join-to-path* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a circle. The circle may end up looking like an ellipse on the screen if the current transformation matrix does not scale x and y uniformly.

center-x The x-coordinate for the center of the circle.

center-y The y-coordinate for the center of the circle.

radius The radius of the circle.

Of the listed keyword options, **:inner-radius**, **:start-angle**, **:end-angle**, and **:clockwise** are unique to **graphics:draw-circle** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:inner-radius
Specifies the inner radius of a circular ring figure; the default is 0.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 50 50 50 :inner-radius 25))
```

Viewer: Default Viewer

Current Candidates
(FLAVOR:METHOD :DRAW-CIRCLE TV:GRAPHICS-MIXIN)
(FLAVOR:METHOD :DRAW-FILLED-IN-CIRCLE TV:GRAPHI
PRINT-CIRCLE
:BORDER Option to FORMAT-GRAPH-FROM-ROOT
:CLOCKWISE Option to GRAPHICS:DRAW-CIRCLE
:END-ANGLE Option to GRAPHICS:DRAW-CIRCLE
:INNER-RADIUS Option to GRAPHICS:DRAW-CIRCLE
:SHAPE Option to SURROUNDING-OUTPUT-WITH-BORDER
:START-ANGLE Option to GRAPHICS:DRAW-CIRCLE
Drawing Polygons and Circles on Windows
Graphic Output within Tables
GRAPHICS:DRAW-CIRCLE
GRAPHICS:DRAW-CIRCLE-DRIVER
GRAPHICS:DRAW-UNFILLED-CIRCLE-DRIVER
GRAPHICS:TRIANGLE-CIRCUMSCRIBED-CIRCLE
How the Reader Recognizes Lists
Keys Reserved for the User
Notation Conventions for Quoting Characters
Other Basic Facilities for Graphic Output
Printed Representation of Lists
Special Character Names
SYS:EXPRESSION

Bookmarks
▶ GRAPHICS:DRAW-CIRCLE Function

Commands
▶Show Candidates (word(s) [default "underline character"]) circles
▶ *Show Documentation GRAPHICS:DRAW-CIRCLE*
▶

Show Candidates | Help
Show Documentation | Select Viewer
Show Overview | Reselect Candidates
Show Table of Contents | Read Private Document

Figure 1-2: System Documentation for the DRAW-CIRCLE Function

The DOCUMENTEXAMINER provides information about the calling sequence of the DRAW-CIRCLE function and illustrates the effects of some of the arguments. Programmers must trace down other relevant documentation and information on combining functions needed to solve their programming task. Also, programmers first have to request and select among possible topics from those shown in the pane in the upper right.

(FLAVOR:METHOD :BACKGROUND-GRAY DW:MARGIN-LABEL)
(FLAVOR:METHOD :BOX DW:MARGIN-LABEL)
(FLAVOR:METHOD :BOX-THICKNESS DW:MARGIN-LABEL)
(FLAVOR:METHOD :CENTERED-P DW:MARGIN-LABEL)
(FLAVOR:METHOD :CHARACTER-STYLE DW:MARGIN-LABEL)
(FLAVOR:METHOD :DELAYED-SET-LABEL DW:MARGIN-MIXIN)
(FLAVOR:METHOD :DELAYED-SET-LABEL TV:DELAYED-REDISPLAY-LABEL-MIXIN)
(FLAVOR:METHOD :EXTEND-BOX-P DW:MARGIN-LABEL)
(FLAVOR:METHOD :LABEL TV:CHOOSE-VARIABLE-VALUES)
(FLAVOR:METHOD :LABEL TV:LABEL-MIXIN)
(FLAVOR:METHOD :LABEL TV:MENU)
(FLAVOR:METHOD :LABEL-SIZE TV:LABEL-MIXIN)
(FLAVOR:METHOD :MARGIN DW:MARGIN-LABEL)
(FLAVOR:METHOD :MARGIN-COMPONENTS DW:MARGIN-MIXIN)
(FLAVOR:METHOD :SET-LABEL DW:MARGIN-MIXIN)
(FLAVOR:METHOD :SET-LABEL TV:LABEL-MIXIN)
(FLAVOR:METHOD :SET-LABEL TV:MENU)
(FLAVOR:METHOD :STRING DW:MARGIN-LABEL)
(FLAVOR:METHOD :STYLE DW:MARGIN-LABEL)
(FLAVOR:METHOD :UPDATE-LABEL DW:MARGIN-MIXIN)
(FLAVOR:METHOD :UPDATE-LABEL TV:DELAYED-REDISPLAY-LABEL-MIXIN)
:LABEL
:LABEL Option to DW:ACCEPTING-VALUES
:LABEL-ALIGNMENT
:LABEL-PANE Option to DW:DEFINE-PROGRAM-FRAMEWORK
:LABEL-POSITION
:LABEL-SEPARATOR-LINE
:LABEL-SEPARATOR-LINE-THICKNESS
:PANES Option to DW:DEFINE-PROGRAM-FRAMEWORK

Basic Text Formatting Commands and Environments
CLOS:GENERIC-LABELS
Disk Restore FEP Command
DW:MARGIN-LABEL
Edit Disk Label Command
Editing the Disk Label
FEP File Types
FLET, LABELS, and MACROLET Special Forms
L (Kbd) Zmail Command
Label Command
LABELS
Labels on Volumes
Overview of Window Flavors and Messages
Set Disk Label FEP Command
Show Disk Label FEP Command
SI:EDIT-FEP-LABEL
SI:READ-FEP-LABEL
SI:WRITE-FEP-LABEL
TV:DELAYED-REDISPLAY-LABEL-MIXIN
TV:LABEL-MIXIN
TV:MOMENTARY-MENU Example 3: Centered Label and Use
TV:TOP-BOX-LABEL-MIXIN
TV:TOP-LABEL-MIXIN
Window Labels
Window Margins, Borders, and Labels

Figure 1-3: Candidate Documentation Topics for Keyword ''labels''

Given the keyword ''labels,'' the DOCUMENTEXAMINER suggests these topics of the system documentation. An experienced programmer should be able to guess from the names that these topics deal with labels on screen windows, disk labels, and the labels function for defining new functions —nothing to do with drawing labels on graphs.

1.2 Examples in Programming

It is hardly a daring hypothesis that examples provide a useful aid to understanding. Examples are in use all around us as a means of explaining things in the world, and programming is no exception. What C programmer does not know the ''hello world'' example [Kernighan, Ritchie 78, pp. 5-6]? The original user manual for Pascal [Jensen, Wirth 74] also begins with a program example and these words from the authors:

> Much of the following text assumes the reader has a minimal grasp of computer terminology and a 'feeling' for the structure of a program. The purpose of this section is to spark that intuition.

Examples take advantage of peoples' intuition, allowing them to use an artifact even without a full understanding of that artifact. With a little background knowledge, much mileage can be gained from examples [Lewis 88a].

The value of examples was clear in my previous work experience consulting for users of a FORTRAN graphics library called DISSPLA [ISSCO 81]. The documentation for the DISSPLA library was organized into sections of features: one section on basic plotting, one section on using fonts, another section 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. I eventually 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.

This use of examples fits programming particularly well: in programming, it is not necessarily the ''why'' of something that is important, but the ''how.'' Once the ''how'' is determined, the example can provide a starting point for reuse. The benefit of the plots accompanying the code examples cannot be ignored. They provide a visual inventory or table of contents of the features that the program example implements. Further, the context of the example and the particular features that are used can support further thinking and refinement of an envisioned solution.

Explainer

Code

```
(defun cyclic-group-100 nil
  (let ((2pi (* 2.0 pi)) (pi/2 (/ pi 2.0)) (radius 30.0))
    (graphics:with-room-for-graphics (*standard-output* 200)
      (graphics:with-graphics-translation (*standard-output* 150
        (graphics:with-graphics-scale (*standard-output* 2.25)
          (graphics:draw-circle 0 0 radius :filled nil)
          (do* ((theta-incr (/ 2pi 10.0))
                (x)
                (y)
                (theta-list (list 0.0
                                  theta-incr
                                  (* 2.0 theta-incr)
                                  (* 3.0 theta-incr)
                                  (- 2pi (* 2.0 theta-incr))
                                  (- 2pi theta-incr))
                            (cdr theta-list))
                (label-list '("e = 0" "1" "2" "3" "98" "99")
                            (cdr label-list))
                (theta (car theta-list) (car theta-list))
```

Diagram

plot stories → coordinate space (screen area, translation, scaling); circle; labels stories (x ticks, number labels) ... (less) ... (more)

Example Output

3, 98, 2, 99, 1, e = 0

Explanation Dialog

Story – (Tell-Story) – Modulo-Addition Perspective
This plot is a visualization of addition modulo 100. Adding 1 to 1 yields 2. Adding 1 to 98 yields 99. But adding 2 to 98 yields 0 and adding 2 to 99 yields 1. In general, adding two numbers i and j yields (i + j) mod 100. The circle reinforces the idea that the values wrap around. ... (less)

Story – (Tell-Story) – Cyclic-Operations Perspective
This plot is a good way to visualize operations in a cyclic group. ... (more)

Plot – (Tell-How) – Plot-Features Perspective
The plot consists of a coordinate space, a circle, and labels. The coordinate space consists of a screen area, a translation, and a scaling. Labels consists of x ticks and a number labels. ... (less)

Plot – (Tell-How) – Lisp Concept Perspective
The plot consists of a function definition. ... (more)

phrase "labels" = loop = story = labels

SUBJECT: a phrase
A. Highlight On
D. Diagram
E. Text How
F. Text Why
G. Text Story

Typein Commands
Command: Highlight On "Phrase"
Command: Highlight Off "Phrase"
Command:

Figure 1-4: Explainer's Presentation of the Cyclic Group Example

Examples are the key to accessing software information in EXPLAINER. The "Cyclic Group Example" is presented to support programmers working out the Clock Task. The programmer using EXPLAINER is given different views of an example and sometimes different perspectives within those views. Fragments of LISP code, parts of the example picture output, nodes of the diagram schematic, and phrases within the text views may be selected for querying information using the menu shown at the bottom-middle of the screen.

1.3 An Example-Based Approach to Design

The preceding portrayal of examples can be described as a design process consisting of these phases: specification, location, comprehension, and construction (see Figure 1-5). Specification does not necessarily imply a description in a formal language, but does assume that some initial statement of a problem exists. This statement is the foundation for the retrieval of a related example or examples from a catalog of examples. The examples illustrate components such as function calls to system libraries that could support a solution to the problem. Eliciting and reviewing the explanation of the example helps to confirm the relevance of the example and to identify the components that may be extracted from the example and applied to the construction of a solution. Once an initial prototype is constructed, it may be improved upon. Improvements may be derived from studying the examples retrieved so far, or by augmenting the initial description and searching for other, potentially more elaborate examples.

The diagram in Figure 1-5 should not be interpreted as firmly compartmentalizing the process of design. This particular separation represents major needs that are addressed by separate research efforts. Additional cognitive and system issues are investigated in these efforts. The many arrows in the diagram indicate the tight integration required of system components involved with these individual aspects of design. Some of the corresponding system components are described in Chapter 3. The work of this dissertation centers on the comprehension of examples once they are retrieved. Specifically, programmers using EXPLAINER should be helped to recognize when certain components apply to a situation, what results they can provide, and how they might be combined and adapted in a solution.

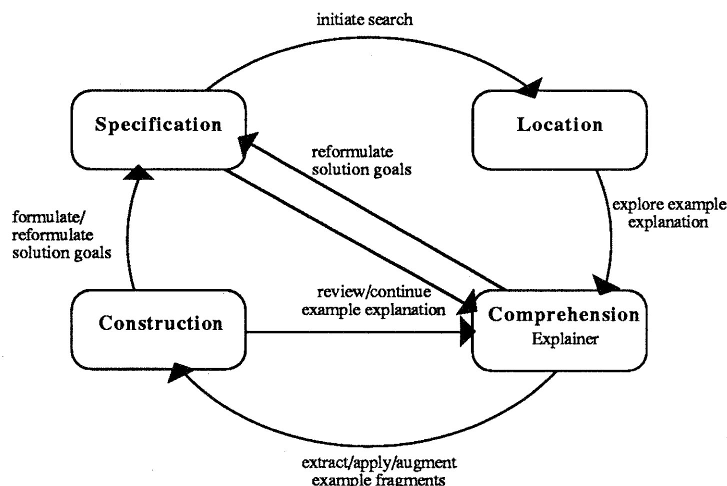


Figure 1-5: An Example-Based Design Process

1.4 The Explainer System

The EXPLAINER system framework exploits these properties of examples:

- examples provide visual clues;
- examples index "how to" information; and
- examples organize information relevant to a common theme.

For the clock programming task, the Cyclic Group Example is provided as in Figure 1-4. From the picture in the example output pane (lower left of the figure), a programmer can see that the example includes features of drawing a circle and placing labels around the circle. Through interactions supported by the pop-up command menu (bottom middle of the figure), the programmer can tie these features to specific parts of program code.

The EXPLAINER approach exploits examples as a way of indexing into software information. It allows programmers a direct way of exploring and answering questions about features illustrated in an example. Examples both demonstrate one way a feature could be coded and provide code material to reuse. The specific context provided by examples helps programmers refine their goals or problem task. Explanations provided by EXPLAINER clarify implementation and design details.

In contrast, consider the on-line documentation system of the Symbolics, the DOCUMENTEXAMINER (see Figure 1-2). This documentation does show an example, but it makes no attempt to describe the example. For instance, the programmer has to know that the "with-room-for-graphics" function is used to set up a plotting space within which the "draw-line" function can be used. Understanding the example must come through understanding the code and independently looking up other functions. The documentation uses an example to illustrate general principles but does not help show the interrelationship of these principles (or corresponding functions), nor does it take advantage of the possibility of an example to serve as a way to organize and index related information.

Examples can also provide additional context for the programmer. In Figure 1-4, not only are descriptions presented in terms of graphics features such as coordinate space, circle, and labels, but also in terms of modulo addition and operations in cyclic groups. Sometimes these ancillary *perspectives* will also match a programmer's task and be a further help in building an analogy from a problem to the example and consequently to a programmed solution. At a minimum, programmers can assume that a graphics feature and a programming-language perspective will be present.

Later chapters elaborate how programmers using the EXPLAINER program are supported in accessing the information in the example (Chapter 3), how that information is represented and interpreted with multiple perspectives (Chapter 4), and how programmers made use of the EXPLAINER tool for solving the clock task with the Cyclic Group Example (Chapter 6).

1.5 Evaluation

The initial idea and conception of the EXPLAINER tool was stimulated by the author's work experience and by general observations of programmers and systems, as described above. The actual construction of the EXPLAINER tool and its evaluation took place in three stages: preliminary observations of programmers, leading to the development of an initial prototype tool; informal observations of students working with the tool, leading to refinements and preparation for testing; and, after further development of the system, evaluation through a formal experiment.

In the first observations, programmers were observed in solving simple programming tasks with the assistance of a human consultant (Chapter 5). The purpose of these observations was to develop an understanding of the kinds of questions and information that programmers would need to know while seeking to understand examples. These observations helped to guide the development of an initial system. They also indicated that the proposed tool would have to fit with programmers' usual patterns of working or transcend these patterns in supportive directions. The strongest work pattern observed was that of prototyping.

With the initial implementation completed, further observations were carried out with students using the tool to help with a class homework assignment (Chapter 5). These served the dual purpose of getting feedback on the actual implementation and of testing the system on a relatively large task. The observations helped confirm that the tool as implemented could support comprehension of program examples. They also led to further refinement of the tool and supported preparations for a formal evaluation.

Finally, the tool was formally tested by comparing the performance of programmers using the fully operational EXPLAINER tool to the performance of programmers working with less powerful tools, primarily the Symbolics DOCUMENTEXAMINER (Chapter 6). The most striking result from this experiment was that subjects using the EXPLAINER tool proceeded to solve a programming problem in a more directed and controlled fashion than did subjects in the other conditions, including those that had the DOCUMENTEXAMINER for a help tool. The EXPLAINER tool seemed to level out the performance of a group in contrast to the wide variation in performance often observed among programmers [Egan 91].

The trend of directness provides a starting point for examining the effectiveness of the EXPLAINER tool and approach over the two rounds of observations and the final, formal evaluation. Specifically, the tool not only fits into but actually supports the prototyping behavior commonly used by programmers; it helps programmers work more effectively; and it provides a surprising payoff for the initial investment of start-up costs (Chapter 7). These initial results support further development in the application domain of programming and suggest additional, related research in the areas of problem solving and learning.

1.6 Reader's Guide

The research reported in this dissertation can appeal to several audiences:

- researchers in the human-computer interface community, especially those interested in the notion of examples as an interface technique;
- researchers who build tools to support programming, software design, and reuse;
- researchers interested in explanation techniques, especially as they are implemented in a specific domain;
- researchers interested in knowledge representation, especially in models that emphasize multiple representation perspectives and structural and declarative simplicity.

The rest of this dissertation is organized as follows. Chapter 2 develops ideas from different research disciplines into a conceptual framework that identifies issues addressed by the tool. The specific approach and prototype tool are described in Chapter 3. The implementation of the tool, both the representation of knowledge and the interface that interprets the knowledge, are discussed in Chapter 4 and a programming scenario illustrates the use of the system. Chapters 5, 6, and 7 examine the effect of the example-based tool on programmers' behavior. Chapter 8 looks at some related work not discussed previously. Chapter 9 summarizes this dissertation's work and discusses future directions.

2. Conceptual Framework for Example-Based Design

2.1 Overview

Research in human computer interaction draws on many disciplines. The EXPLAINER work integrates research results from the study of design, problem solving, knowledge representation, and explanation, and applies them in the domain of software engineering. The discussion of research below defines the culture or community of ideas that have influenced the EXPLAINER approach. A general theme is how the human designer can be supported in problem solving on the computer. These ideas have led to the development of many research efforts within our *Human-Computer Communication Group (HCC Group)* at the University of Colorado.

2.2 Problems in Software Engineering

The short history of software engineering is filled with tools and methodologies for programming. One of the most promising developments is software reuse [Standish 84; Tracz 88; Biggerstaff, Perlis 89]. Software reuse offers not only the obvious potential for repeated use of existing, debugged code, but also —and perhaps more importantly—the opportunity for programmers to structure their designs around proven domain-level abstractions. Simon states the potential more generally: "complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not [Simon 81, p. 209]."

The formulation of the notion of reuse in software is generally attributed to McIllroy in 1968 [McIlroy 76]:

> Software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abbetted by the existence of families of structural shapes, screws or resistors [p. 89].
>
> A comprehensive and physically condensed document like the Sears-Roebuck catalogue is what I would like to have for my own were I purchasing components [p. 95].

McIllroy's idea of software "catalogues" is generally realized today in the form of subroutine libraries. For example, the *Guide to Available Mathematical Software* [NBS 84] for use at the National Institute of Standards and Technology classifies over two thousand functions for numerical computation from over fifteen separate commercial and public libraries. The growing movement toward object-oriented programming has extended the notion of reuse to components (objects) that can be reused directly and incrementally refined [IEEE Software 87; Fischer, Lemke, Rathke 87].

However, the potential of software reuse has yet to be fully realized. As with many innovations, software reuse brings with it new problems [Fischer, Henninger, Redmiles 91; Curtis 89]. Some of the obstacles are *artifact-centered*, problems of creating and storing modules. These fall in the domains of software engineering and programming language design. By contrast, the HCC Group has focussed on problems that are *human-centered* [Fischer et al. 92a; Fischer, Henninger, Redmiles 91; Reeves 91]:

1. users do not have *well-formed* goals and plans,
2. users do not know about the *existence* of components,
3. users do not know how to *access* components,
4. users do not know *when* to use components,
5. users do not understand the *results* that components produce for them, and
6. users cannot combine, adapt, and modify components according to their *specific* needs.

The EXPLAINER work focusses on the fourth and fifth problems: when to use components and how to understand their results. However, the work is conceived within an approach that addresses all of these issues, both with theory and systems (see Chapter 3). To help programmers determine when to use existing software components, the example-based approach of EXPLAINER provides a context in which

programmers can see situations in which those components are applied. To help programmers understand the components, EXPLAINER shows working combinations. The notion of a software "catalogue" is thus enriched to encompass a catalog of previously solved programming situations, in which a programmer can explore available components and their application.

2.3 Examples to Support Programming

The value of enriching the concept of the software catalog to include illustrative situations is supported by cognitive studies of examples. These studies encompass research in learning [Pirolli, Anderson 85; Lewis 88a; Chi et al. 89; Papert 80], problem solving [Kintsch, Greeno 85], text comprehension [Dijk, Kintsch 83; Kintsch, Dijk 78; Pennington 87], information retrieval [Tou et al. 82], and design [Schoen 83; Fischer, Nakakoji 91a].

Pirolli and Anderson [1985] observed that examples played a crucial role for students learning LISP concepts. Lewis observed that people can use and generalize from even a single example, relying on heuristics and some background knowledge [Lewis 88a; Lewis 88b]. Chi and colleagues observed that students can learn well from examples, provided they are careful to explain the examples to themselves [Chi et al. 89].

Several terms have been ascribed to these notions of working with and learning from examples. Schoen's *reflection in action* requires designers to be working in the context of a problem to experience feedback on how to proceed [Schoen 83]. The situation is said to *talk back* to the designer. This bears a resemblance to the ideas of *retrieval by reformulation* [Tou et al. 82], FischerNieper-Lemke1989] or *query by example* [Ullman 82]: users revise queries based on cues from examples, intermediate retrievals. Papert has a similar notion, that learning requires an *object to think with* [Papert 80]. Likewise, Suchman puts forth the need for *situated action*: detailed steps in carrying out tasks can never be fully specified beforehand, but will be dictated while carrying out the task [Suchman 87].

What are the salient features of examples that make them so valuable in problem solving and learning? Lewis observed these four [Lewis 87, p. 254]:

1. examples show abstract principles in use;
2. they constrain the interpretations of principles;
3. they afford extension by analogy without full understanding; and
4. they provide details that then need not be recalled or rediscovered.

But, both the work of Lewis [Lewis 88b] and Chi and colleagues [Chi et al. 89] point to the necessity of explanation and some background knowledge in interpreting the examples. Additionally, the problem solver needs to be aware that the example is relevant to the current task [Gick, Holyoak 80].

In the work of Kintsch, Greeno, and vanDijk [Kintsch, Greeno 85; Dijk, Kintsch 83; Kintsch, Dijk 78], a person's background knowledge is known as the *situation model.* The way in which information about a problem description must be organized to allow a solution is termed the *problem model.* Mapping elements of a problem description into a problem model is accomplished through a *problem-solving strategy,* which is part of the situation model. The organizing structure of the text of the problem description is called its *macrostructure* and the constituent pieces are its *microstructure.*

With respect to software tasks, Kintsch and Fischer replaced the problem model with the *system model* [Fischer 87a; Fischer, Nieper 87, p. 9]. In this scheme, the situation model is the problem solver's informal understanding of a software task, including solution goals and problem-solving strategies. The system model is the formal organization of the problem into a sequence of operators the computer can interpret for execution. A valid system model allows the problem solver to produce software, a program, implementing the starting task. Elements of the task description (task microstructure) have been mapped into elements of a program code (program microstructure) with a systems organization (program macrostructure) (see also [Pennington 87; Majidi, Redmiles 91]).

The concept of a program macrostructure is equivalent to the notion of a programming plan [Adelson, Soloway 88; Soloway, Ehrlich 84; Rist 89]. Plans in this sense focus on the decomposition of solution

goals into implementation operators. Plans are subsumed by examples. As Suchman demonstrates, abstract plans are insufficient to capture the complexities of real situations [Suchman 87]; plans are merely resources. In software engineering terms, specifications can never be complete [Fischer, Reeves 92]. Examples, on the other hand, have the advantage of having accommodated the details of real situations. They have a kind of authority or validity: they once succeeded.

Examples are also needed to constrain the variety of plans that are possible in modern computing environments. *High functionality computer systems* [Fischer 87b] provide programmers with thousands of functions across many separate component libraries. These different subsystems and functions have often been created under different circumstances and for different needs. The end result is often function overlap and sometimes function incompatibility. In cases where different conceptual models are involved, examples may be the most practical means of explanation. An example can bypass abstractions and demonstrate the effect of functions directly in the domain of the current task. In the case of function overlap and potential incompatibility, examples constrain the programmer's search space: an example shows one possible way to make a set of functions work together.

2.4 Representing and Explaining Examples

The successful use of examples in problem solving relies on the ability of people to build analogies. Gentner [1983] developed the theory of *structure-mapping* to help explain the analogical process. ''Salient'' concepts and relationships about an example constitute a *system of knowledge*. The analogy is defined and measured by the number of concepts and relationships that correspond between the systems of knowledge from a *base* to a *target* example.

The ''base'' would be a supplied example to work from. The ''target'' would be the task a programmer needs to solve. The development of the structure mapping is the problem-solving strategy and depends on the programmer's situation model. First, the situation model provides the background knowledge needed to identify appropriate concepts and relationships [Lewis 88a]. Second, what is ''salient'' about the base and the target depends on the programmer's goals for solving the problem. As noted by Fischer in [Fischer, Nieper 87, p. 9], these goals may evolve as the problem is worked on.

In graphics programming, the domain on which this research focusses, a general goal is to implement some feature. A programmer may have an informal idea of what this feature is, maybe a ''ring'' or ''tire.'' Examples can illustrate how features such as circles and ellipse may appear, helping the programmer reformulate his or her goals and situation model. Examples can also show how ''analogous'' goals have been decomposed into a system model [Fischer, Henninger, Redmiles 91].

Explanation can help to make this decomposition evident by focussing on the macrostructure of the example program [Soloway et al. 88; Pennington 87; Majidi, Redmiles 91]. Explanations can take on a variety of forms, ranging from traditional textual expositions to graphical depictions of module interrelationships to animated annotations of running programs. A key part of this dissertation is the investigation and evaluation of different forms of explanation for program examples.

As an example of explanations in nontextual form, Rathke explored how spreadsheet forms could be explained through decomposition [Fischer, Rathke 88]. Cells whose values were computed could be opened into graphical networks illustrating dependencies. If cells at this level were also composite, they could be further expanded.

Many other approaches attempt to exhibit program structure visually. Computer-aided software engineering (CASE) tools commonly use diagrams of data flow and calling hierarchies [Fairley 85; Reiss 85; Sodhi 91]. Hierarchical class browsers have become commonplace with the growth of object-oriented programming [Goldberg 84; Rathke 86a]. Meyers surveys many visual programming languages [Myers 86]. Visual tools enhance people's ability to recognize and index into knowledge of the object being portrayed [Larkin, Simon 87]. Hence, an additional incentive to begin the explanation work of this thesis in the graphics domain.

For textual explanation, some tools for computer generation exist [Meteer et al. 87; Mann 85]. Swartout, Moore, Paris, and Bateman have developed techniques for explaining the reasoning of expert systems in a variety of domains [Swartout 83; Moore 87; Bateman, Paris 89]. They have focussed on the need to

provide explanations that are more than rule-firing histories. While these explanation generators are not yet efficient enough to incorporate into interactive system, the motivation for this work emphasizes that explanations based simply on a system's structure are insufficient.

The need to provide alternative *perspectives* or *points of view* has been explored in the visual programming and CASE tools mentioned above. These approaches provide alternative external presentations but the representations still reflect only the structure of the software. Other research starts from the assumption that plans or alternatives have to be supported at the higher level: e.g., general, visual problem-solving environments proposed by [Riekert 87; Riekert 86; Larkin 89]. Providing representations that incorporate alternative perspectives has been a recurring theme in knowledge representation, where alternative descriptors for objects have been provided for different use situations [Bobrow, Winograd 77; Moore, Newell 74; Minsky 75] and have been incorporated in some experimental object-oriented, knowledge representation languages [Rathke 86b].

Another theme in automated explanation is incremental explanation. Many of the object-oriented browsers and the spreadsheet work of Rathke allowed levels of abstraction to be exposed gradually. G.L. Fisher [Fisher 85] explored how an interface could allow users to traverse levels of abstraction by supporting incremental expansion from formal specifications to code statements. Incremental explanation has the advantage that the programmer is not overwhelmed by information structures that are too large to be comprehended without extensive effort [Fischer et al. 90].

2.5 Summary

Modern approaches to programming, including object-oriented approaches, rely heavily on reuse from libraries of functions. However, several problems prevent programmers from using such libraries as effectively as might be possible. Two of these problems are the need to determine when to use components from the library and the need to understand the effects of those components. To address these issues, the notion of libraries as catalogs is enriched to encompass a notion of a catalog of examples illustrating the application and combination of software components. The use of examples is supported by cognitive studies that demonstrate how examples can help programmers solve tasks working by analogy: programmers can apply parts of the macro- and microstructures of a program example to the solution of their current task, helping them develop a working system model. Various forms of explanation can play an important role in helping programmers to understand examples and form productive analogies.

3. Approach of the EXPLAINER System

3.1 An Example-Based Model of Design

The experience with the example-based consulting model mentioned in the introduction, reinforced by the observations and issues discussed in the previous chapter, led to an example-based model of design, in which human problem-solving abilities are augmented by computer tools. The augmentation approach [Engelbart 88; Fischer, Nakakoji 92] has grown in popularity in the field of software engineering [Fischer et al. 92a; White 91; Green et al. 86] due to the problems of implementing a fully automated approach [Rich, Waters 88]. A popular approach to augmentation is to provide a knowledge-based assistant [White 91; Rich, Waters 90]. In the example-based approach, the computer's role as assistant is to provide examples appropriate to a programmer's current task and to support the programmer in developing a solution through analogy to the example.

The computer's role as an assistant leads to a model of programming as a *cooperative problem-solving* activity [Fischer 89; Fischer 90]. Each party brings its own strengths to the job of completing the current task. In the example-based approach, the computer provides a store of existing program examples and facilities for making these examples and the information they embody available; the human brings some background knowledge, along with abilities for interpreting the current task and building analogies to the examples provided.

As envisioned in this approach, examples provided by the computer consist of a description of the task solved by the example, the program code that implements the solution, and explanation relating the goals of the example to both the macro- and microstructure of the program code. Explanations may be non-textual, e.g. taking the form of component diagrams. The illustration of how goals of the example task were implemented in program features would suggest to the programmer ways to accomplish and perhaps embellish the solution of the current task [Schoen 83; Papert 80], thus supporting the programmer's development of the situation model. The programmer's problem-solving strategy is to work by analogy [Lewis 87; Lewis 88a]. Parts of the macro- and microstructure of the program example may be adapted in the solution of the current task. They support the programmer's development of the system model.

To interpret examples, programmers need some background knowledge. The basic background knowledge for understanding LISP code is rather simple: functions are applied to arguments and functions are composed to make programs. For graphics programs written in LISP, results of functions correspond to graphic features. Computer support can help identify the roles of functions and arguments, and it can highlight the relationship between functions and graphic features. The programmer will, however, need to judge the relevance of features to the current task.

A model of computer assistance following the approach outlined above is diagramed in Figure 3-1. The ovals identify phases, or states, in the example-based design process; the arcs identify programmer activities. The phases and activities will be detailed below. Briefly, programmers begin by formulating some solution goals for their current task in an initial *specification*. These goals form the basis for a query to a catalog of program examples. When the programmers are satisfied with the initial specification, they can initiate a search in the catalog. The search leads to the *location* of one or more examples relevant to the stated goals. By exploring the examples through the explanations provided, programmers can judge the relevance of the examples. During this phase of *comprehension*, the programmers may decide to articulate different or more precise goals. If an example meets expectations, the programmers can extract parts of the example for inclusion in a solution to the current task. During *construction* of the solution, additional examples may be sought to accommodate needs not met by the examples provided to that point.

The model has been greatly influenced by the architecture of the *multifaceted design environment* developed by Fischer and colleagues [Fischer, McCall, Morch 89; Fischer, Nakakoji 91a; Fischer et al. 92b]. Part of the philosophy of that approach is that different aspects of design are best served by different system components. In this sense, the approach leads more to an environment for design than to a single "assistant." The different components are expected to be tightly integrated. In the example-based model of design, the high degree of integration is indicated by the activity arcs.

The phases of specification and location, comprehension, and construction correspond to separate system

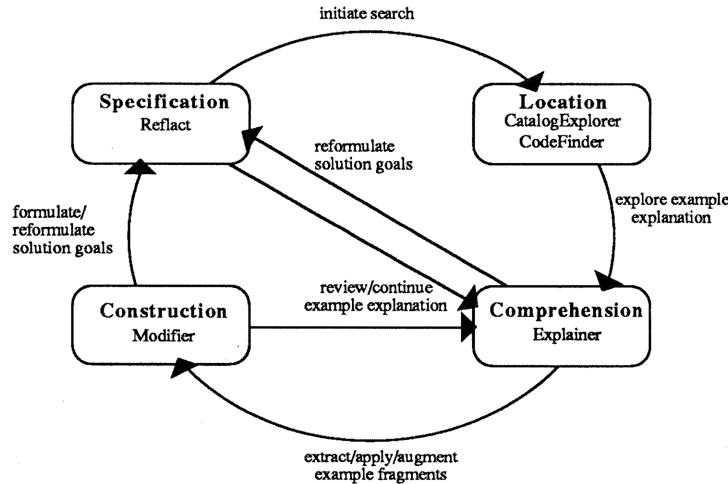


Figure 3-1: Processes and Supporting System Components in Example-Based Design

components being developed in the HCC Group at the University of Colorado. The EXPLAINER component, which is the focus of this dissertation, supports the comprehension phase. Nakakoji and Henninger have researched and developed systems for specification and location [Fischer, Nakakoji 91b; Henninger 91]. Girgensohn has developed theory and systems for the modification of existing components [Girgensohn 92] and Lemke and Morch have developed systems for construction [Lemke 89; Morch 88]. The integration of the components of specification, location, comprehension, and construction has been examined (see [Fischer et al. 92a]). The rest of this chapter describes how these components interact to produce a productive design environment for a programmer working on a task. The discussion focusses primarily on the contributions of EXPLAINER, with brief technical summaries of the other systems.

3.2 Specification and Location

Nakakoji developed the CATALOGEXPLORER system to support designers in specifying and retrieving

Questions	Answers
Current object: GRAPHIC-PROGRAMMING DEF	Current issue: What is a purpose of the illustration? Current answers: illustrate the correlation of the two values;
? What should be the design of graphic programming? ? What is a purpose of this graph? ? How many objects do you have to illustrate? ! **What is a purpose of the illustration?** ? Which type of graph do you want to use? ? Which type of line do you want to use? ? Does the graph have a title? ? Does the graph have a subtitle? ? Where to put the title?	•**illustrate the correlation of the two values** •trace changes over time

(a) IBIS-Style Questionnaire

Specify the factor of importance for each specified item.	Least									Most
What is a purpose of this graph? To plot values	☐	☐	☐	☐	☐	☐	☐	☐	☐	☒
How many objects do you have to illustrate? Two	☐	☐	☐	☐	☐	☐	☐	☐	☐	☒
What is a purpose of the illustration? illustrate the correlation between the two values	☐	☐	☐	☐	☐	☐	☐	☐	☐	☒
Which type of graph do you want to use? A simple line graph	☐	☐	☐	☒	☐	☐	☐	☐	☐	☐
Does the graph have a title? Yes	☐	☐	☐	☐	☐	☐	☐	☒	☐	☐
Does the graph have a subtitle? No	☐	☒	☐	☐	☐	☐	☐	☐	☐	☐
Do It ☐	Abort ☐									

(b) Weighting Sheet

Figure 3-2: Specification Component of CATALOGEXPLORER
(reproduced by permission from [Fischer et al. 92a])

Designers can specify their design requirements in the form of a questionnaire (a). The left window provides designers with questions. By clicking one of them, the right window provides possible answers to select. After the specification, designers have to weight the importance of each specified item (b).

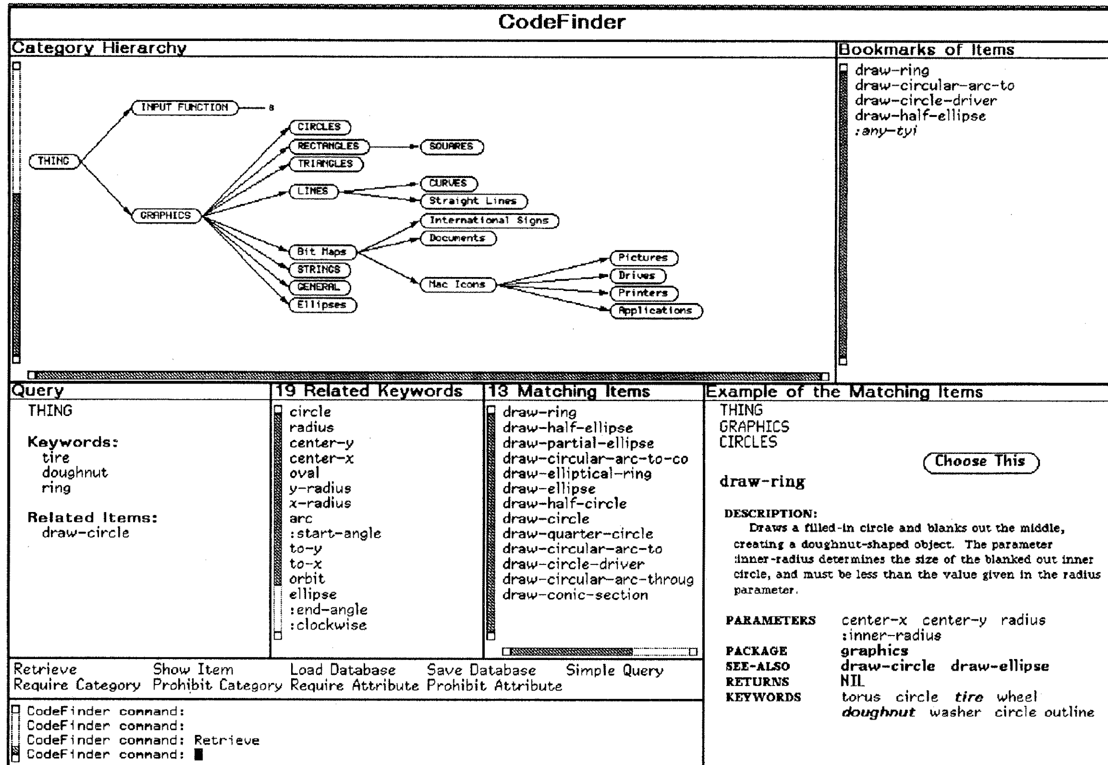


Figure 3-3: CODEFINDER User Interface
(reproduced by permission from [Fischer, Henninger, Redmiles 91])

The CODEFINDER user interface is based on HELGON [Fischer, Nieper-Lemke 89]. The Category Hierarchy window displays a graphical hierarchy of the information space. The Query pane shows the current query. The query parts combine to retrieve the items in the Matching Items pane. The Example of the Matching Items pane shows the full entry for one item in the information space. The Bookmarks pane holds a history of the objects. The Matching Items pane shows all items matching the current query, by order of relevance to the query. The Related Keywords pane shows keywords retrieved by the query. Any of these keywords can be added to the query. The remaining panes allow users to specify commands by mouse or keyboard.

catalog examples based on problem requirements (see [Fischer, Nakakoji 92] for details). The specification elicited from the programmer partially identifies the designer's goals for a solution. From this partial specification, CATALOGEXPLORER locates relevant example programs in the catalog base. It infers relevance by using *specification-linking rules* that are dynamically derived from a *domain argumentation base*. The rules map a high-level abstract specification to a set of condition rules over program constructs.

The domain argumentation base is based on the Issue-Based Information System (IBIS) structure [Conklin, Begeman 88; Fischer et al. 91; Kunz, Rittel 70]. It consists of issues, answers (possibly contrasting alternatives), and arguments for the answers. This information is accumulated by recording design rationale for design decisions made in previous design sessions. The specification component (see Figure 3-2a) is an issue-base hypertext system that allows designers to add such information and to articulate positions in the issue base as a requirements-specification activity.

Each of the answers and arguments in the specification component is associated with one of several predefined *domain distinctions*. For example, in graphics programming, domain distinctions include types of graphs (e.g., line graph, bar graph, circular graph), emphases (e.g., transitions, comparisons), nature of data (e.g., continuous, discrete), and design features (e.g., dividing line, x-axis/y-axis, center of a circle).

In the IBIS structure, answers to issues are encouraged by pro-arguments and discouraged by contra-

arguments. Pro-arguments lead to positive dependencies, ''X implies Y.'' Contra-arguments imply negative dependencies, ''X implies not Y.'' When programmers specify a domain distinction in their design requirements by selecting a certain answer, the system finds other answers that have arguments tied to the same domain distinction. The domain distinctions of the found answers are used to define specification-linking rules. The system uses the specification-linking rules to find all example programs in the catalog that have some of the required domain distinctions.

CATALOGEXPLORER orders the found examples according to computed appropriateness values. When programmers articulate specification choices, they are asked to assign a weight to each choice, indicating its degree of importance (see Figure 3-2b). The appropriateness of an example in terms of a set of specification items is defined as the weighted sum of the satisfied conditions provided by the related specification-linking rules. By seeing the effects of changing the degree of importance in the ordered catalog examples, programmers can make trade-offs among specification items.

Another approach to locating examples and other software components for reuse was taken by Henninger in the CODEFINDER system (see [Fischer, Henninger, Redmiles 91] and [Henninger 91] for details). This system's interface (see Figure 3-3) uses the retrieval by reformulation paradigm [Fischer, Nieper-Lemke 89; Tou et al. 82] to help programmers incrementally build queries by critiquing intermediate retrievals. The actual search for components uses an algorithm based on associative, spreading activation [Bein, Smolenksy 88; Mozer 84]. Retrieval by reformulation improves the spreading activation paradigm by allowing incremental refinement of a query. The spreading activation algorithm supports the query process by using soft constraints to overcome the need for the programmer to specify precise vocabulary.

3.3 Comprehension

The effect of the location phase is to present programmers with a program example ''relevant'' to their current task. The goals of the EXPLAINER tool are to help programmers quickly judge if the example is indeed relevant and, if it is, to determine what aspects of the example can be adapted to the current task. EXPLAINER supports these goals by allowing its users to explore different *perspectives* and *views* of the example (see Figure 3-4).

3.3.1 Perspectives, Views, and Explanations

Perspectives are descriptions of an object organized around a specific theme. The explanation dialog pane in Figure 3-4 (lower right) shows four perspectives of the Cyclic Group Example: perspectives of modulo arithmetic, cyclic group, plot features, and LISP programming. The different perspectives focus on different concepts and relationships organizing the concepts. This notion of perspectives is consistent with its use in the KRL system [Bobrow, Winograd 77], the Merlin system [Moore, Newell 74], the ObjTalk language [Rathke 86b], and points of view described in Minsky's theory of frames [Minsky 75].

In EXPLAINER, perspectives can be presented in different formats, or views. The plot-features perspective is not only presented as a textual description in the explanation dialog pane, but also as a hierarchical diagram in the diagram pane, Figure 3-4 (upper right). EXPLAINER supports four different kinds of views: code listing, example output (picture), diagram, and text. These are typical views supporting program documentation. Different perspectives have default view types associated with them. By default, concepts and relations of a programming language perspective would be viewed as a code listing, although the same information could be diagramed or presented in text.

Initially, minimal information is presented on the EXPLAINER screen [Fischer et al. 90]. The initial views are meant to indicate the variety of information available. In addition to the sample execution picture and the code listing, a diagram of the plot-features perspective and a text description from the domain perspective are provided. This latter provides a description of what task the example was meant to solve. The philosophy is to invert the normal order of exposition followed by computer documentation: examples consisting of code and sample execution are foremost, elaborations interesting the programmers are requested incrementally.

Programmers can expand descriptions and diagrams by clicking on the ''(more)'' prompts. In Figure 3-4, some text descriptions and branches of the tree have already been expanded this way. Additional infor-

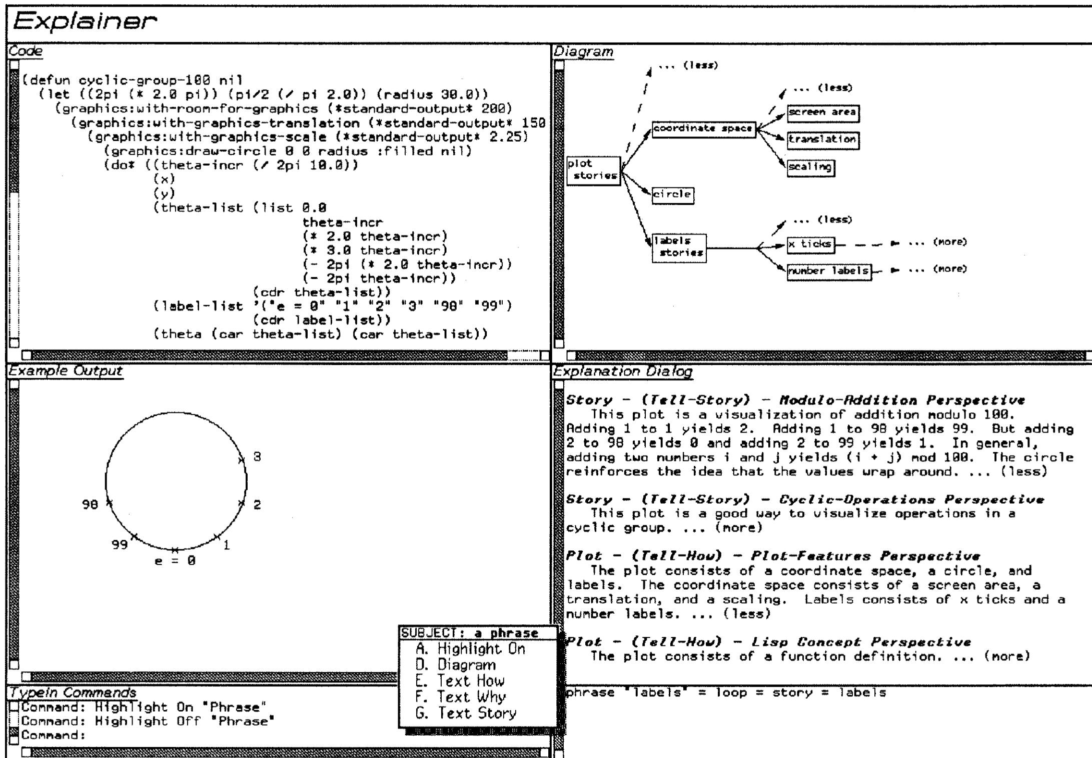


Figure 3-4: Explainer's Presentation of the Cyclic Group Example (Repeated from Figure 1-4)

mation can be retrieved through the interaction menu as programmers explore the example. The interaction menu, Figure 3-4 (middle bottom), is a pop-up menu that appears when screen fragments in different views are selected with the mouse. The menu actions apply equally to all fragments on the screen. Parts of the code can be selected and diagramed (menu action "Diagram"). Diagrams can be described in text (menu action "Text How"). Correspondences between concepts in different perspectives and views can be highlighted (menu action "Highlight On"). Because highlighting occurs in all views, this allows the programmer to point to a graphic feature, highlight it, and observe which section of code implements that feature.

Thus the programmers' understanding of the task drives the exploration. As the programmers' understanding of the task evolves, previously expanded views can be eliminated and new ones explored. When a location tool, such as CATALOGEXPLORER, is more fully developed, it will be integrated with EXPLAINER to provide more information about programmers' tasks and to allow EXPLAINER to suggest starting points in the different views and perspectives. At a minimum, the concepts the location tool uses to identify an example as relevant would be presented and highlighted.

Thus, in EXPLAINER, *explanation* is defined to be the presentation of concepts and relations within a perspective of an example, and the illustration of the interrelationship of the concepts between perspectives and views. Such explanations show how the macro- and microstructure of a program code implements graphic features according to a given system model, and how these structures relate to the task the example originally solved.

3.3.2 Judging Relevance

The programmer's need to make quick judgements of relevance is supported in part by the graphical domain. By checking the example output pane, programmers can determine many of the graphical features the example includes. The Cyclic Group Example presented in Figure 3-4 illustrates the graphic features of a circle with labels around its perimeter. Less visible distinctions could be checked in the diagram pane where the initial diagram decomposes the example by graphic features. The fact that a coordinate space is created is identified here.

The default paragraphs in the explanation dialog pane always present a description of the original task solved by the example. The description is in the original domain terms, although it may be followed by a more generalized description. For judging relevance, these initial paragraphs show the programmer if there is any additional overlap of the example with the current task, e.g., similar domains as well as similar graph distinctions.

Programmers using EXPLAINER must have enough background knowledge to appreciate the information EXPLAINER presents. As mentioned above, they need a basic understanding of graphics, enough to recognize graphic features, and they need an understanding of the problem, enough to suspect what graphic features are needed for the solution of their current task.

As part of their background knowledge, the programmers also need to know how to access information using the EXPLAINER interface. To make experimental testing of the program more practical, great effort was put into simplifying this interface. An informal, cognitive walkthrough[1] of the interface led to several changes, including reduction of the principle interaction menu from over 20 items to 5, revisions of wording to make it more easily recognizable in the context of a task, and more equal distribution of the screen to the different panes. In the end, about 20 minutes was needed to train graduate computer science students to adequately use the interface (see Chapter 6). Thus, the background knowledge needed to operate the EXPLAINER interface is not a major issue for its users.

3.3.3 Building Analogy

Once determining that an example may indeed be relevant, programmers need to explore the example to identify which aspects might be applied in implementing a solution to the current task. The analogy programmers need to develop is one that relates a supplied example to the current task. As discussed in the previous chapter, Genter [Gentner 83] interpreted analogy to be the identification of ''systems of knowledge'' and the development of a mapping from the base to the target system. Later she explored the effect of the task in determining the salient features of objects when identifying the systems of knowledge and developing the mapping [Clement, Gentner 91]. In EXPLAINER, perspectives play the role of Gentner's systems of knowledge and identification of the task solution goals is achieved during the specification and location phases.

The EXPLAINER approach, then, presupposes that a common perspective exists between programmers' tasks and the supplied examples. In a sense, the existence of a common perspective is guaranteed; minimally, it is found in the graphics features perspective. If an example has been retrieved, say using the CATALOGEXPLORER tool described above, then that example has been identified to have ''domain distinctions'' (graphic features) that are relevant to how the programmers have identified their solution goals. The CATALOGEXPLORER helps programmers identify these goals through the use of argumentation and then automatically maps them onto combinations of graphic features.

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 constructs. Some of the actions for exploring an example were mentioned above. Most of the features are illustrated in Figure 3-5, which shows the state of the EXPLAINER interface at the end of one test user's programming session.[2] The programmer was using the Cyclic Group Example (presented by EXPLAINER in the figure) to develop a solution to the clock task (Figure 1-1). The screen shows that the programmer had

- 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'');

---

[1]This informal walkthrough was greatly facilitated by John Rieman, one of the researchers working on the method [Lewis et al. 90].

[2]This test user was a participant in the formal experiment discussed in Chapter 6.

Figure 3-5: Final EXPLAINER Screen in a Test Session (Subject #1)

- 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 which 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.

The EXPLAINER tool supports a programmer's system model by presenting views and descriptions of the macrostructure of the program and by illustrating how the macrostructure maps into the microstructure of LISP elements. By presenting the macrostructure from a graphics-features perspective, combined with information developed in the specification and location phase, EXPLAINER enables the programmer to develop an analogy between the example and the solution goals and then apply parts of the macro- and microstructure of the example to the solution of the current task.

In some cases, the task solved by the the example might match closely the current task. In such cases, the mapping from the task perspective to the graphics-features perspective can support programmers. For instance, if instead of the clock task, programmers were given the task to illustrate arithmetic modulo 10, they could expand descriptions from the modulo-arithmetic perspective and simply identify the mapping from there to the programming level. The study of examples where task descriptions match closely as compared to nonmatching tasks (e.g., the Cyclic Group Example as compared to the clock task) is an area of future research.

3.4 Construction

Construction includes two types of activities: adaptation of example components to produce a new

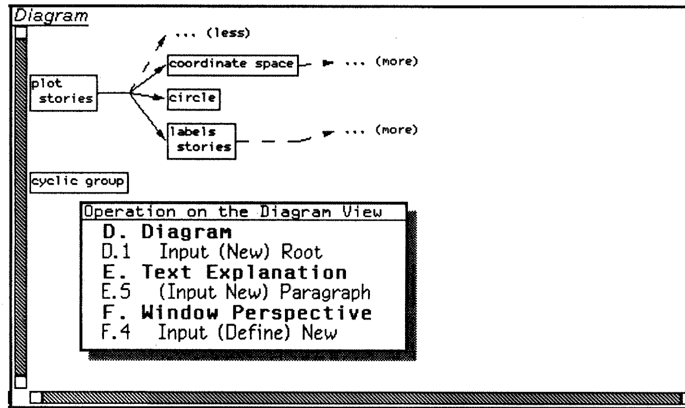


Figure 3-6: Adding a New Concept

The new concept "cyclic group" is added through the concept input menu.

Explainer

Code

```
(graphics:with-graphics-translation (*standard-output* 150
  (graphics:with-graphics-scale (*standard-output* 2.25)
    (graphics:draw-circle 0 0 radius :filled nil)
    (do* ((theta-incr (/ 2pi 10.0))
          (x)
          (y)
          (theta-list (list 0.0
                            theta-incr
                            (* 2.0 theta-incr)
                            (* 3.0 theta-incr)
                            (- 2pi (* 2.0 theta-incr))
                            (- 2pi theta-incr))
                      (cdr theta-list))
          (label-list '("e = 0" "1" "2" "3" "98" "99")
                      (cdr label-list))
          (theta (car theta-list) (car theta-list))
          (label (car label-list) (car label-list))
          (x-attachment))
         ((null theta-list) nil)
      (setq x (* radius (cos (- theta pi/2)))
```

Diagram: ... (less); coordinate space; ... (more); plot stories; circle; labels stories; ... (more); ... (less); cyclic-group; elements; operator

Example Output: 3; 98; 2; 99; 1; e = 0

Explanation Dialog: Story – (Tell-Story) This plot is a vi... (more); Story – (Tell-Story) This plot is a go... cyclic group. ... (n...

EDIT: a an elements
A. Any Concept
A.2 Equate
A.3 Disassociate
C. Code
D. Diagram
D.1 Input (New) Root
D.3 Input (New) Part
D.4 Delete (Old) Part
D.5 Reorder Part
E. Text Explanation
E.1 (Edit) What
E.3 (Edit) How
E.4 (Edit) Why
E.5 (Edit) Story
F. Window Perspective
F.4 Define (New) Perspective

Typein Commands
Select another concept: #<EXP::GRAPHER-NODE 17285432>
Command: Highlight On "Elements"
Command:

elements = variable name LABEL-LIST = list = label list

Figure 3-7: Adding Subcomponents to a Concept

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.

program, and augmentation of the catalog knowledge base with new examples or additional knowledge about existing examples. The adaptation of components to a new task has to date relied on existing tools. While EXPLAINER helps programmers recognize what components can be applied to the solution of the current task and how, the actual reuse requires cutting and pasting with an existing editor, such as EMACS. However, tools for reuse that maintain knowledge behind components and their combinations are envisioned, based on techniques similar to those used for constructing and annotating new examples as described below.

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 [Girgensohn 92] and some work done by Jeff

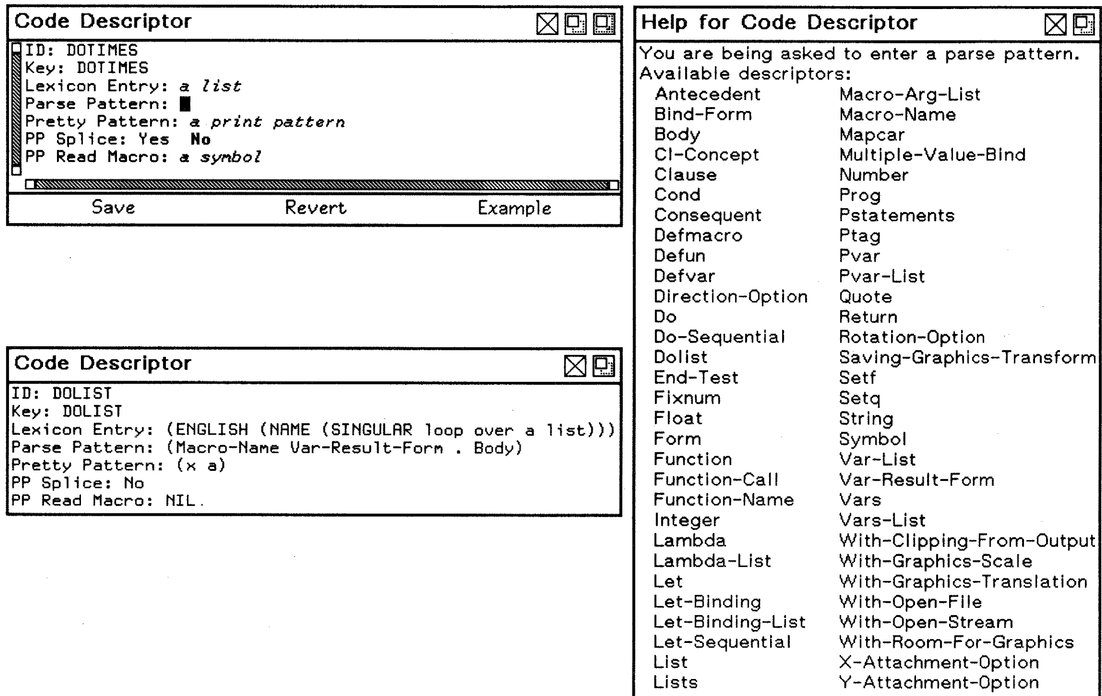


Figure 3-8: Constructing New Objects in EXPLAINER

In order to add knowledge about a LISP construct, a new code descriptor that describes how LISP code can be parsed and printed has to be defined in a property sheet (top-left window). Existing descriptors can serve as examples (bottom-left window). The help window (right) shows a list of the existing descriptors. For each of these descriptors, additional information can be requested.

Hanson and Steve Russell as part of a class project. The input of knowledge about examples is a semi-automated 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, as shown in the detail of the diagram pane in Figure 3-6. Using a concept editing menu, concepts from different perspectives can be equated. For instance, the concept ''elements'' shown in the diagram pane can be equated to the variable ''label-list'' shown in the code pane, Figure 3-7. The object net of concepts associated with a program example is saved together with the example.

Descriptors associated with some concepts can be edited in property sheets (see Figure 3-8). The fields of these sheets have help displays associated with them. The help window displays all possible values, such as all program code descriptors in the system. Descriptions for any object in the help window can be requested.

The construction phase supports both the evolution of individual examples and the evolution of the catalog base. The design environment itself evolves through the introduction of new domain knowledge, for example, adding the new concept ''cyclic group'' or new patterns for parsing program code. An individual example starts with parsed program code and evolves through the addition of perspectives that describe the function of the example.

3.5 Summary

A model of design based on the use of examples has been developed. The model supports the development of the situation and system models by supporting a problem-solving strategy of analogy from examples, allowing opportunities for reformulation of goals and, if necessary, retrieval of additional examples. The model specifically helps programmers with the problems in software engineering discussed in the previous chapter (see also [Fischer, Henninger, Redmiles 91]):

- specification, and the ability to reformulate, helps programmers develop their goals and plans;
- specification and location help programmers learn about the existence of components and help them access examples;
- the examples help programmers see how to apply and combine components and understand what results are possible;
- construction supports the input of new examples and suggests ways programmers can be supported in modifying examples themselves.

Examples are represented and explained through multiple perspectives and views of these perspectives. The intent is to make it possible for programmers to recognize and identify aspects of the example program's macro- and microstructure for application to the solution of the current task. The visual domain simplifies the recognition of these structures as well as determining two minimally required perspectives: the graphics-features perspective and the LISP perspective. Users of the EXPLAINER tool may expand these and other perspectives and see the mapping between them. The presentation of information in EXPLAINER inverts the normal order of exposition followed by computer documentation: examples consisting of code and sample execution are presented first, and programmers request incremental elaboration of points of interest or difficulty.

4. Implementation—Representation and Interpretation

4.1 Overview

This chapter explains in detail the implementation of the EXPLAINER tool. It concludes the first half of the dissertation, which has been devoted to the development of EXPLAINER. The second half of the dissertation focusses on the evaluation of the tool in use. In this chapter, the goals of the approach are reviewed with attention to the aspects that guide the implementation. The manner in which these goals are realized in the implementation is illustrated in a scenario, presented from the point of view of a programmer using EXPLAINER to solve a task. The scenario also illustrates the functionality that the interface needs to support and the knowledge that lies behind it in the example. Next, the overall architecture of the tool is described. The chapter concludes with discussions of the representation of knowledge in EXPLAINER and the algorithms for interpreting that knowledge.

4.2 System Goals

As discussed in the last chapter, the EXPLAINER tool is part of an approach to programming based on problem solving by analogy to previously worked-out program examples. Analogies can be developed along many perspectives, but minimally, an analogy exists from the perspective of graphic features that are common to the example and the task. From this point of view, the examples illustrate how desired graphic features, such as circles and labels, are implemented by software components, such as function calls.

The purpose of the EXPLAINER tool is to help programmers view these different perspectives and their relationships, especially the mapping between the graphics-features perspective and the LISP perspective. By making the macrostructure of the example available along these particular perspectives, EXPLAINER supports programmers' development of analogy and solutions. EXPLAINER uses examples to show what software components are used, how they are applied, and how they are combined in the context of a complete program to achieve certain graphic effects. This kind of help is also suggested by Solloway and colleagues in their call for a documentation theory of *delocalized plans* [Soloway et al. 88].

To support this approach and its goals, an implementation was envisioned in the style of hypermedia systems such as the Symbolics CONCORDIA system [Symbolics 88] on which the DOCUMENTEXAMINER tool was built (see also the Apple's HYPERTALK system [Apple 89] and the VIRTUALNOTEBOOK system [Shipman, Chaney, Gorry 89]). The interface would present multiple views of the information for an example: code listing, sample execution, component diagrams, and text. The same information in one representation perspective would be presentable in many different views. A LISP concept such as the call to the function that draws the circle in the Cyclic Group Example might be 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 viewing the sample perspective information in different external forms accommodates different persons' preferences. A person preferring text over diagrams would be able to access the same information. The multiple views also provides reinforcement of new concepts through redundancy.

A difference from other hypermedia implementations was the higher granularity of indexing. Almost all items presented on the screen would be sensitive and be equally accessible for command actions. That is, all the same command actions would be available independent of the view (code, sample output, text, or diagram). This granularity would be made possible by generating some information automatically instead of requiring all of it to be authored. For example, instead of simply being able to retrieve a picture illustrating the output of an example, pieces of the picture would also be sensitive.

4.3 Users' Scenario

The purpose of the scenario which follows is to show, in sequence, what features would used by a programmer trying to use EXPLAINER to help with a specific task. The value of specific features is

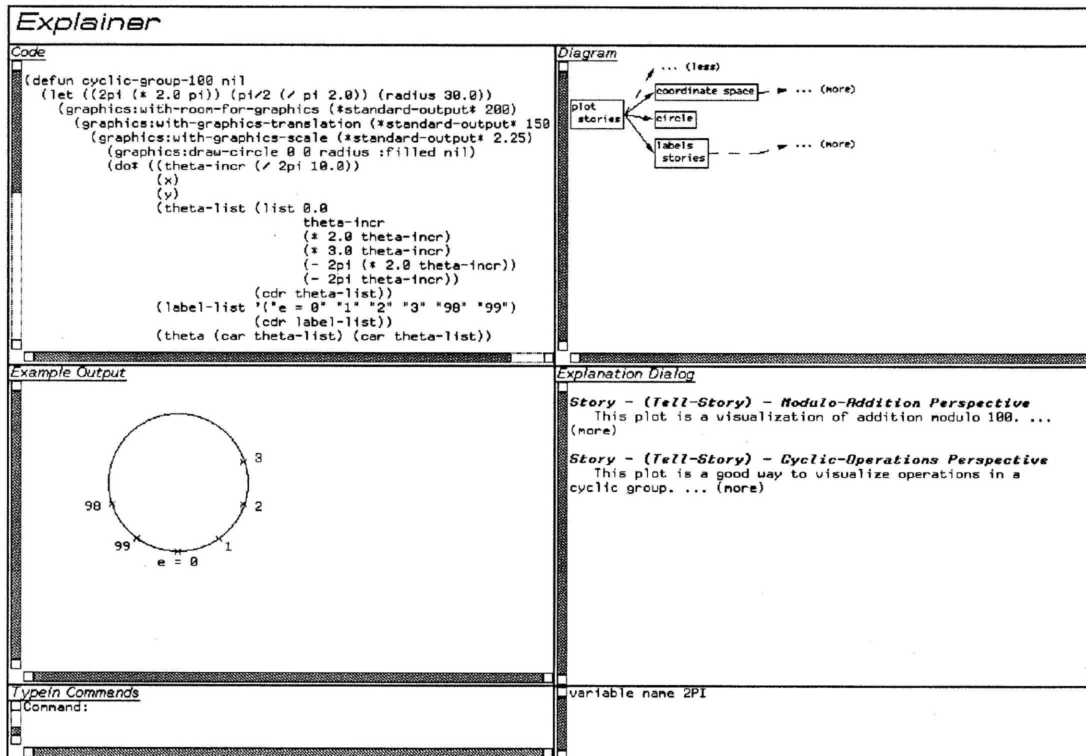
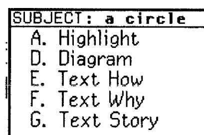
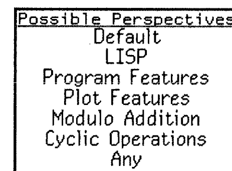


Figure 4-1: EXPLAINER: Initial Screen





(a) (b)

Figure 4-2: Pop-up Menus in EXPLAINER

emphasized by describing the programmer's actions, what he or she would see on the EXPLAINER screen, and by summarizing what knowledge the programmer might recognize that could be applied to the solution of the task. These latter knowledge summaries are underlined as an aid to the reader.

The task is the Clock Programming Task given in Figure 1-1. The scenario begins at the point when EXPLAINER would be engaged, i.e., after the programmer has specified and located an example from the catalog. A retrieval tool such as the CATALOGEXPLORER or CODEFINDER might have been employed.

After studying the programming task, the programmer may have articulated, or been helped by one of the location tools to articulate, the need for an example illustrating components for drawing a circle with labels. From the initial EXPLAINER screen Figure 4-1, the programmer can recognize that example is relevant by looking at the example output (picture) pane in the lower left of the screen. It shows two principle features in common with the task: drawing a circle and labels.

The programmer might want to expand one of the initial stories in the text pane in the lower right of the screen in Figure 4-1. These initial text descriptions always summarize the original task solved by the example. In this case, the example task of visualizing modulo addition has essentially no overlap with the current task of drawing a clock face (see expanded description in Figure 4-3). The programmer can ignore further references to this cyclic-group perspective.

The programmer might next want to find out how the circle is implemented. By clicking on the circle, the programmer activates the temporary pop-up menu shown in Figure 4-2a. In Figure 4-3, the result of

Explainer

Code

```
(defun cyclic-group-100 nil
  (let ((2pi (* 2.0 pi)) (pi/2 (/ pi 2.0)) (radius 30.0))
    (graphics:with-room-for-graphics (*standard-output* 200)
      (graphics:with-graphics-translation (*standard-output* 150
        (graphics:with-graphics-scale (*standard-output* 2.25)

          (do* ((theta-incr (/ 2pi 10.0))
                (x)
                (y)
                (theta-list (list 0.0
                                  theta-incr
                                  (* 2.0 theta-incr)
                                  (* 3.0 theta-incr)
                                  (- 2pi (* 2.0 theta-incr))
                                  (- 2pi theta-incr))
                            (cdr theta-list))
                (label-list '("e = 0" "1" "2" "3" "98" "99")
                            (cdr label-list))
                (theta (car theta-list) (car theta-list))
```

Diagram

... (less)
coordinate space ... (more)
plot stories
labels stories ... (more)

Example Output

Explanation Dialog

Story - (Tell-Story) - Modulo-Addition Perspective
This plot is a visualization of addition modulo 100. Adding 1 to 1 yields 2. Adding 1 to 98 yields 99. But adding 2 to 98 yields 0 and adding 2 to 99 yields 1. In general, adding two numbers i and j yields (i + j) mod 100. The reinforces the idea that the values wrap around. ... (less)

Story - (Tell-Story) - Cyclic-Operations Perspective
This plot is a good way to visualize operations in a cyclic group. ... (more)

Typein Commands

Command:

Figure 4-3: EXPLAINER: Expanded Objects

selecting the "Highlight" command is to highlight several items: the function call in the code listing that implements the circle, the role of the circle concept in the overall program structure illustrated in the diagram pane, and even the role of the circle in the task description. For efficiency on the Symbolics, the highlighting of the circle in the example output pane was approximated by a square. The programmer now knows what function creates the circle and can experiment with varying different parameters.

From the highlighting action, the programmer's attention may be drawn to the diagram pane. If the programmer was not already aware of its value, he or she might now try exploiting it. The diagram shown in Figure 4-1, shows the structure of the program from the graphics-features perspective. It is another view of the information of the picture. However, certain features which are not visible in the picture, such as the "coordinate space," can be identified in the diagram view. If the programmer were to expand the node for "coordinate space" (expansion not shown in the figures), he or she would see that it consists of three subnodes presenting the concepts of "screen area," "translation," and "scaling." The programmer might think these are important but choose to ignore them for the present and in fact, hide them again from sight.

Instead, the programmer might expand the "labels" concept, as in the upper right of Figure 4-4. Again, something that might not have been clear in the picture is that the labels consist of both "x ticks" and "number labels." The programmer can now decide that "x ticks" are not needed on the clock face and concentrate on finding out how the "number labels" are implemented.

Expanding the "number labels" concept, the programmer sees two subcomponents, "position" and "number," in the upper right of Figure 4-4. Noticing that the node for "position" is annotated with the label "story," the programmer knows that he or she may retrieve a textual description written by the example's author. Clicking on the node brings up the command pop-up menu from Figure 4-2a. Selecting the command "Text Story" causes the perspectives pop-up menu in Figure 4-2b to appear. Selecting the program-features perspective yields the text in the middle of the text pane in Figure 4-4. The programmer learns that the coordinates are initially specified as radians and then converted to rectangular coordinates. This may help the programmer's interpretation of other elements of the program code, such as the uses of "pi" and "pi/2."

By clicking on the "number labels" node in the diagram and selecting "Highlight" again from the

Figure 4-4: EXPLAINER: Final Screen
(Repeated from Figure 3-5)

pop-up menu, the programmer can see the related concepts in the different views (also shown in Figure 4-4). In particular, the programmer can see that the labels depend on two different functions in the LISP code: an assignment converting radian values to rectangular coordinates and a function drawing the number label at the converted position. Equivalent information can be presented in text in the explanation dialog pane in the lower right of Figure 4-4. The repetition of the information can reassure the programmer about the operations implementing, in this case, the position for the label. The programmer now knows where labels are drawn and how coordinates are represented in the program. He or she may try variations on these function calls and contributing components. During the formal evaluation of EXPLAINER (see Chapter 6), a common action at this point was for a programmer, working with a copy of the program code, to change the form "(+ radius 5)" to "(- radius 5)" in order to place the labels inside the circle perimeter.

Thus, a programmer can investigate the elements (microstructure) of a program and their role in the programming plan (macrostructure), particularly from the perspective of graphic features. The different views give the programmer flexibility in indexing the different information. The mapping between concepts in the different perspectives is evident in the highlighting between views; this contributes to the programmer's knowledge of how graphic features are implemented. The ability to view the same concepts in different views gives a redundancy that can reassure a programmer about new information he or she is investigating. The programmer can expand or contract presentations of information as he or she chooses to avoid an overload of data irrelevant to the current task. EXPLAINER provides a programmer with information needed in implementing a solution, assuming his or her ability to analogize from the example to the current task.

4.4 Implementation Architecture

EXPLAINER is implemented in the CLOS object-oriented programming system and the COMMON LISP programming language on the Symbolics LISP-machine and uses the presentation and graphics substrate specific to the Symbolics. In CLOS, the terminology is that *methods* implement algorithms organized around *classes* of objects. The generic classes are instantiated into specific *instances* which form the data

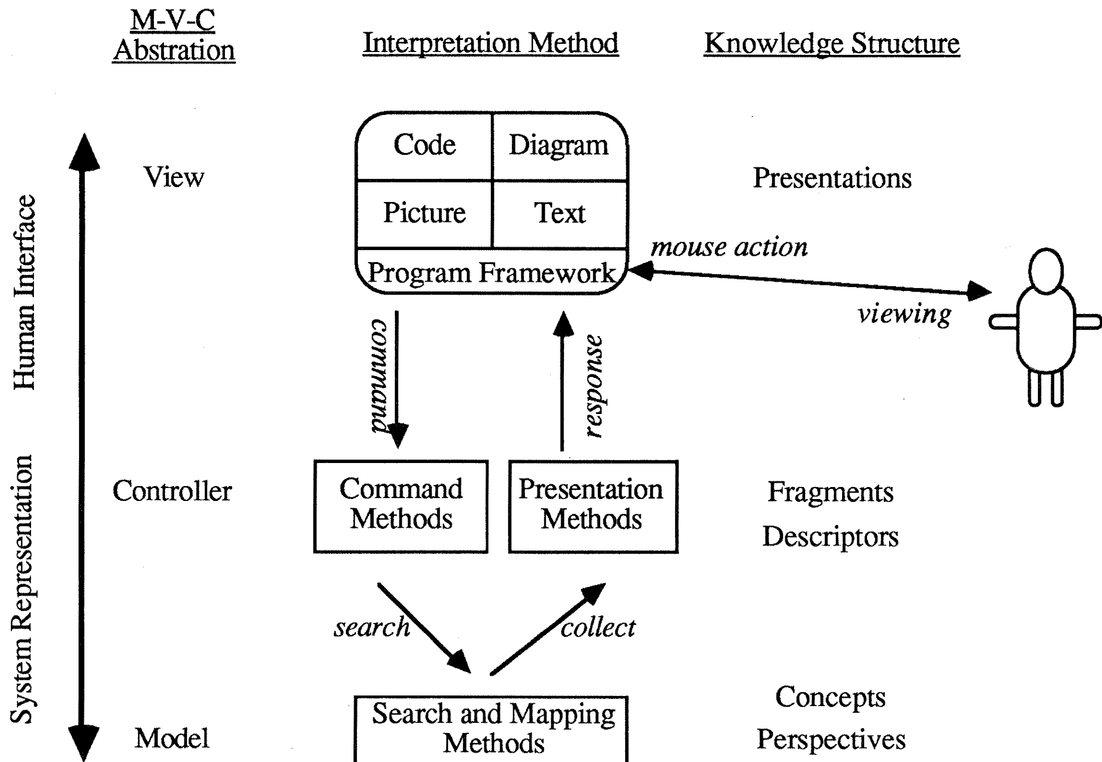


Figure 4-5: EXPLAINER Architecture: Theory, Methods, Knowledge, and Data Flow

for a program. See [Keene 89] and [Bobrow et al. 88] for the details on CLOS. On the Symbolics, the presentation substrate makes possible the overall program framework, including the different panes for the views. It also supports the mouse interactions and command menu applied to objects in the program framework. The Symbolics presentation substrate is very similar to the new CLIM standard [Symbolics 91]. The graphics substrate provides the ordinary facilities for drawing graphic primitives such as lines and circles, though the specific functions and underlying model are unique to the Symbolics.

Figure 4-5 provides an overview of the architecture of the EXPLAINER tool according to three aspects: (i) its abstract design, (ii) the methods interpreting user actions and manipulating system knowledge, and (iii) the knowledge structures applicable at different levels of the design and interpretation. The abstract and overall design, illustrated in the first column of the figure, is based on the *model-view-controller (M-V-C)* user interface paradigm [Krasner, Pope 88], in which users' interactions are separated from the representation of knowledge. In EXPLAINER, users interact with a *view* of the example. Their interactions are interpreted by *controller* procedures which manipulate the example knowledge *model* and further affect the view.

In EXPLAINER, as illustrated in the second column of the Figure 4-5, the user interacts with the program framework, seeing the different views of the example and applying command actions to concepts presented in these views by selecting items from the EXPLAINER menu. The commands are implemented by methods that search the example representation for information appropriate to the command, sometimes generating new information in the data model, and then present the collected results back in the program framework.

The objects the user sees and interacts with in the view, as illustrated in the third column of Figure 4-5, are *presentations* supported by the Symbolics presentation substrate. Roughly, they manage bitmap information for the Symbolics console. Presentations belong to the view level of the M-V-C paradigm. EXPLAINER records presentations in *fragments*. Fragments belong to the controller level of the M-V-C paradigm. Their primary function is to maintain a link back to *concepts* that they present from the model level. Different fragment types correspond to different view types: code, example output, diagram, and

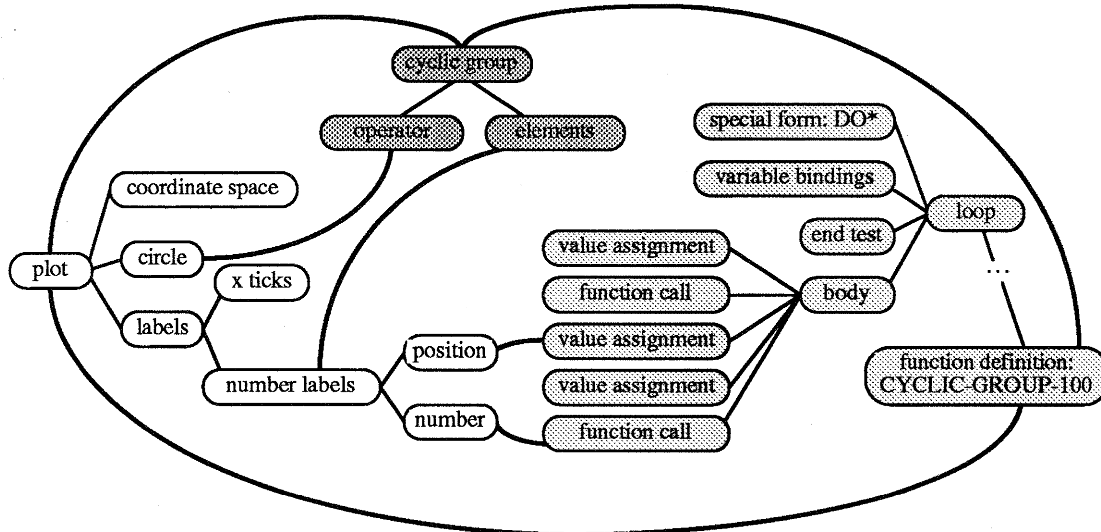


Figure 4-6: Partial Repesentation of an Example

Three perspectives are shown in this figure: graphic features (unshaded ovals), LISP (shaded ovals), and cyclic group (darkest ovals). The straight lines represent components/roles links. The arcs correspond to perspectives links.

text. Many fragments, of the same or different types, can exist for one concept in the example. This is one advantage of the M-V-C paradigm, given the goal that EXPLAINER support multiple views. The actual concepts representing an example exist independently of the classes and methods for presentation. The primary method applied to the representation of example concepts is a search for information to present. The interpretation of concepts for presentation is aided by *descriptors*. Depending on the command action selected by the user, example concepts collected by a command method might be presented in any one of the four views. Descriptors provide information for converting concepts into fragments for a given type of view. For example, descriptors provide patterns for forming concepts into a sentence for a text view and for properly indenting in a code view.

The crux of the EXPLAINER system resides in the knowledge behind examples and the way that knowledge is searched by commands. The remainder of the chapter examines in greater detail the representation of examples and parameters for search.

4.5 Representation of Examples

The scenario in this chapter and the discussion in the previous chapter described the characteristics of examples as used by EXPLAINER and showed how knowledge behind the examples supported programmers in solving tasks by analogy. The examples consisted of code listings and sample executions as is usual in programming. This basic information was supplemented by programming plans capturing the macro- and microstructure of the examples from different perspectives (graphic features, LISP, and problem-domain concepts) and the interconnectivity of these perspectives.

The representation of example information in EXPLAINER is accomplished using *semantic networks* [Woods 85; Quillian 85] (see Figure 4-6). In EXPLAINER, nodes in a semantic network correspond to *concepts* in a given *perspective*. As discussed in the previous chapter, a perspective is a theme or point of view under which the example may be described. Three perspectives appeared in the scenario: a programming language perspective of LISP, a graphics-features perspective, and a problem-domain perspective of cyclic group concepts. Concepts in the LISP perspective consisted of function calls and arguments. Concepts in the graphics-features perspective consisted of graphic concepts such as "number labels" and "x ticks." Concepts in the cyclic-group perspective consisted of problem-domain concepts such as "operator" and "elements."

The networks connect nodes through three kinds of links: *components, roles,* and *perspectives.* These

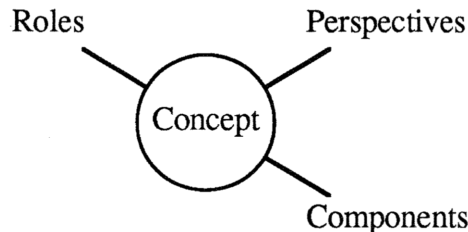


Figure 4-7: Schematic of Concept Representation

links are illustrated in Figure 4-7 and are implemented through slots in a CLOS class for concepts. The components link connects one concept to zero or more concepts that may comprise it. In the Cyclic Group 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 links 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 graphics-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 of individual concepts. Thus, as a whole, an example program is one semantic network that may be interpreted according to various perspectives.

4.6 Interpretation—Parameters for Search

In a sense, all of the implementation of EXPLAINER is devoted to supporting the user interface. The approach outlined in the preceding chapters emphasized the necessity of keeping the human programmer involved in the programming activity; to take responsibility for the interpretation of the programming task, to judge relevance of possible solution information, and ultimately to engineer a solution. EXPLAINER's role is to present relevant and useful information, such as the different perspectives of the macro- and microstructures, in ways that support human programmers. The ''reasoning'' EXPLAINER performs is the search and presentation of information about the example. Though complex and lengthly, the methods for presenting the views of example information and for interpreting menu commands are straightforward and are not described here. However, every command depends on the search method, and since that method has some unique properties, it will be described below.

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 and the more/less ellipsis are essentially means to search for and expose additional concepts in the example's network. Clicking on ''(more)'' in the diagram view exposes additional components of a concept, presented as nodes in the diagram. Applying the ''Text How'' command to a concept in any view exposes that concept's components in a sentence. Using the ''Highlight'' command exposes all the perspectives links from a specified concept by highlighting equivalent concepts in all the views.

As described above, the network representing an example is composed of all the networks in various perspectives. All of the different perspectives are related through some concept. At the least, perspectives are related through a *root* concept which is equivalenced to the main function of the LISP code for that example. Thus, any concept can be reached from any other, though some are distant with respect to the number of intervening links and nodes. For practical purposes, the minimal information has been selected by the author of an example to provide a starting point that is near (within two or three links) of concepts illustrating the graphic features of an example.

Thus, when the user of EXPLAINER selects a command requesting more information, a search begins with a concept that has been presented in one of the views in the interface. The search traverses the links connecting nodes in the network (Figure 4-6) and ends with a related concept or concepts. The target may be immediate concepts on the components link (to respond to a "how" request) or the immediate concepts on the perspectives link (to fulfill a "Highlight" request). However, the search can traverse any kind of link any number of times, although cycles terminate the search in failure. The search proceeds in a breadth-first fashion since it is often necessary to find the shortest path between a source and target.

The search method supports a variety of menu command actions by allowing the target to be any concept, fragment, perspective, or any positive or negative combination of these three. For example, a target of "function-definition" would traverse links from a concept until a concept identified as a "function-definition" was found. A target of "any concept with LISP perspective" would find the closest element in the program microstructure to the starting concept. "Not cyclic-group perspective" would prevent the search from terminating with concepts in that perspective, although concepts and links in that perspective could be traversed to continue the search.

The method allows the search path to be restricted in several general ways. First, the path can be restricted in length to be a minimum or maximum length or to return the immediate (and in breadth-first search, the shortest) path. A maximum length of one would allow the search to look only at immediate neighbors, such as components in the same perspective. Second, the path can be restricted by the kinds of links it traverses. To answer "how" questions, only components links are traversed. Third, the number of times a perspectives link may be crossed can be restricted. This possibility restricts the search in a practical way. While crossing several components and roles links will locate concepts distant from the source, these concepts are not too distant logically since they are in the same perspective. However, several crossings of perspectives and components or roles links will remove the target much further from the original theme. Fourth and finally, the search can be terminated by elapsed time, returning whatever paths reached a target specification within that time. This restriction was necessary out of respect for the user. The tool would be unusable if the commands did not respond after a few seconds. In a sense, since the search is breadth-first, the more relevant information would be identified within a relatively short time frame. Experimentation led to the choice of a four second limit for searches. This limit is perhaps longer than desirable but was necessary due to the speed of the available hardware.

To summarize, EXPLAINER provides access to information about an example. The information is accessed through commands that trigger searches through the semantic network representing the example. The searches are tailored to each particular command for efficiency and semantic appropriateness. The annotated code for the search method is provided in Appendix I.

4.7 Summary

The goals of the implementation were to

- support analogy—by describing examples from different perspectives (LISP programming language, graphics-features, problem-domain);
- support solutions—by highlighting the mapping from different perspectives to one another, and in particular, to the implementation perspective of the LISP programming language;
- support human access to information—by providing many views (code, sample execution, diagram, text) to the information in the different perspectives.

These goals are supported by an implementation organized according to the model-view-controller user interface paradigm, which induces a modular structure that separates support for users' interactions from the representation of knowledge. This organization supports the use of multiple views of an example's information. The information supporting an example is implemented as a semantic net combining concepts from subnets of different perspectives. The user accesses the information in these networks through commands that initiate tailored searches for appropriate information in the net.

5. Initial Observations

5.1 Informal Observations

During the development of EXPLAINER, two sets of informal observations were carried out to guide the development of the system and to prepare for the formal evaluation described in Chapter 6. The first round of observations was done before the system building began; a human consultant provided explanations of examples. These observations confirmed some proposed ideas concerning the example-based working model and suggested issues to be considered in building a prototype. The second round of observations tested an early prototype of the system and provided further feedback for system development as well as suggesting steps needed for formal testing with human subjects.

5.2 Observations with a Human Consultant

The primary purpose of the observations with a human consultant was to get feedback on how people would work with examples under the circumstances envisioned by the tool. Specifically, could programmers solve a graphics programming task by working from a related example? The programmers were allowed to ask the human consultant any questions they wanted. The consultant answered all questions about the example and about LISP but none directed at the attempted solutions. For example, questions about what a function in the example did were answered, but questions about whether the solution was correct were not. At issue were what sorts of questions would be asked by the programmers, whether the programmers could draw an analogy between the given example and the program needed to solve the task, and how having an example might affect the solution. An ancillary purpose of the observations was to get feedback that could help plan future observations and formal experiments.

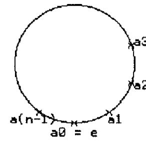
Four programmers participated in the observations. All were Ph.D. students in computer science who knew LISP, and except for one, did not have a extensive experience programming graphics. Three were given a programming task and an example which they were told would help with the solution to the programming task. The fourth, the participant with graphics experience and the greatest experience overall, chose ahead of time as a challenge to work without the example. One task all programmers received was the clock task, as previously shown in Figure 1-1, however no graphic was provided in the task description. The example provided for this task (Figure 5-1a) was an earlier version of the Cyclic Group Example, previously shown in Figure 1-4.

The programmers had the code for the example in an editor buffer and on paper. The first three chose to modify a copy of the code to produce the solution to the clock task. The programmers declared when a solution was complete. A consultant sat next to the programmers to answer any questions they might want to ask. Subjects were told at the beginning of a session and reminded one to two times during the session that they could ask questions.

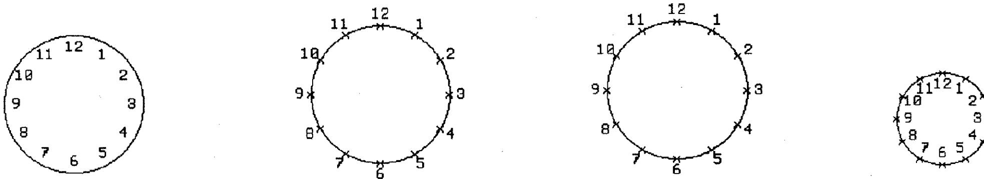
The programmers took between 15 and 60 minutes to create a solution they declared to be complete. They tried between one and eight intermediate solutions. The final output of the four programs along with the output of the starting example is shown in Figure 5-1.

What is interesting to note is that subjects had different visions of what the goal was, even for the simple clock task. This suggested the need to provide a picture of the goal, which was done for the formal experiments described in the next chapter. The goal picture helps to answer a universally asked question during the early observations: "Am I done?" Making the task clearer through the picture encourages, though does not guarantee, that the programmers will attempt to accomplish the same work while solving the task, simplifying comparisons among participants.

The reason for not providing the picture in the early round of observations was that one goal was to learn in what ways examples might affect solutions. Would programmers accept the structure and particular application of functions selected in an example or simply use the example as a context for understanding functions which they later applied in different ways? As evidenced by the final output pictures, the former tended to be the case. This result supported a hypothesis that examples could provide not only an illustration of function calls but also a structure for combining the functions according to a particular task.



(a) Appearance of the Starting Example



(b) Appearance of the Four Subjects' Final Outputs

Figure 5-1: Example and Final Outputs for Clock Task

This support from the example in terms of structure was emphasized when the ''challenge'' programmer attempted to solve the clock task. This programmer had seen the final pictures produced by the other subjects and was therefore somewhat indirectly influenced by the example, but he did not see any of the example code. His initial solutions included a filled-in disc and numerals in reverse order around the perimeter of the disc. The solid disc was produced because he failed to supply a ''fill'' parameter to the circle function. The reversed numerals were the result of a missing coordinate system initialization. These details show the value of a complete example in helping programmers recognize how to apply and combine systems functions.

In these observations, the programmers did not exhibit any difficulty in drawing an analogy between the supplied example and a potential solution to the task, and more specifically, they were able to identify what aspects of the example to modify in order to produce a solution. The ability of programmers to build working analogies is a fundamental assumption with the EXPLAINER approach. It is believed that letting the EXPLAINER users know that an example is related to a potential solution is critical [Gick, Holyoak 80] (see Section 3.3.3).

One of the principal issues of interest in these observations was what sorts of questions programmers would ask about the examples. However, relatively few questions were asked compared to what was expected. One explanation is that these subjects, being Ph.D. students in computer science, were expert users and could have themselves served as the consultant. This idea is supported by an informal observation that the less advanced the student was in degree status, the more questions were asked. The most experienced and advanced of the programmers was the one who did not even want the example code.

For future studies, the implication was that programmers should be selected from a more typical population, i.e., not advanced Ph.D. students, to better reflect the anticipated end users, computer programmer professionals.

A second and more interesting phenomenon was that these students were very independent; as advanced Ph.D. students, they were used to carrying out their own research as well as prototyping systems on their own. This independence suggests a more general barrier to any new tool: established work patterns. New tools often come with assumptions about an overall work framework. Being used to working independently and being used to not having a consultant is a pattern that the EXPLAINER tool will need to overcome. Usually, sufficient interest can be stimulated for the duration and circumstances of a short experiment. Additionally, the tool can contribute to the breaking of usual work patterns in two ways: being a new object to focus on and being interesting and useful in itself. Since EXPLAINER's assumptions about an overall work framework are based on experience in a real world environment, it is believed that the greatest barrier in this round of observations was the background of the programmers.

Programmer: I forget—how does the let statement work?

(a)

Programmer: What's the format for printing reals - f?

(b)

Programmer: I didn't know what this part did.
Consultant: It moves the label locations away from the circle edge.
Programmer: Oh, then I could put the numbers [clock numerals] inside [the circle].

(c)

Programmer: I guess I don't know what this part does.
Consultant: It draws these x's on the edge.
Programmer: So I presume I want that.

(d)

Figure 5-2: Dialog Excerpts from Observations with Human Consultant

Of the questions that were asked by the programmers, two categories could be established: questions about the example and questions about the solution (see Figure 5-2). Questions about the example were the ones most hoped for. It was anticipated that this kind of design information could be represented. Another result supportive of the EXPLAINER approach was that questions were directed at different perspectives. There were questions about both LISP (Figure 5-2a and b) and graphics (Figure 5-2c and d). Questions about the solution pose difficulties for the EXPLAINER tool by itself. One question asked universally was whether the solution was correct (in the sense of meeting the specification in the task description). This question of course is unsolvable by the computer being equivalent to the "Halting Problem." However, as the excerpt from the dialog in Figure 5-2c illustrates, explanations from examples can help programmers reformulate their own specifications.

Another occurrence was that some programmers made programming errors. An error made by one subject was not adding enough numerals. This mistake was easily recognized. An error by another subject was not incrementing a loop counter. Again, this kind of error is beyond the scope of the EXPLAINER tool.

These errors make it clear that the EXPLAINER tool is not a solution to all problems of programming. While it can suggest solutions and help programmers know when and how to apply system functions, it still needs to be used in conjunction with syntax-directed editors, debugging aides, and specification tools. Informally, the questions about the solution may be called "questions that can't" be answered.

Another type of question may be called "questions that weren't" asked. This category represents needs for information that could be provided (information about the example) but that was not explicitly solicited by the programmers. An example given already is the question in Figure 5-2c. This bit of protocol came out during a post-session interview. The programmer may have benefited by asking a question but did not explicitly articulate one while solving the task. Another instance occurred when a programmer spoke out loud: "I need to draw a circle—put text on the screen—tough part is locating points on screen." The question that the consultant could have answered would have been, "Which code draws the circle, the labels, locates the points?" Finally, all three programmers who worked from the example traced through the code in the buffer with the mouse pointer, occasionally pausing on some code fragment. Presumedly, this fragment was some code that required further study, and perhaps the programmer could have benefited from some explanation at this point. All the above unspoken questions about the example led to the development of the crude suggestion mechanism in EXPLAINER. It provides some brief information about objects the mouse points to, but perhaps more importantly, it serves as a constant reminder that the users may ask questions.

A final observation about the work pattern, which had not been anticipated, was that many questions were not articulated because the programmers were interested in getting one prototype done and then later may

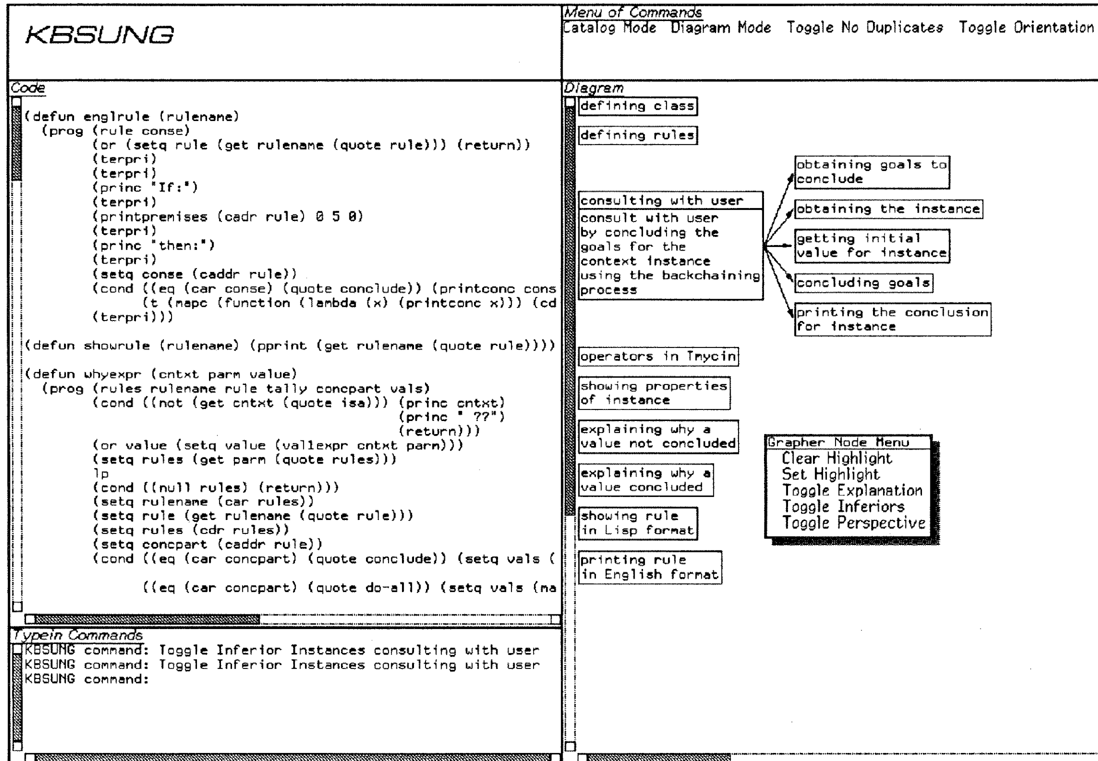


Figure 5-3: The EXPLAINER Prototype as used in the Student Observations (reproduced by permission from [Majidi, Redmiles 91])

(The EXPLAINER tool was at one time called KBSUNG for Knowledge-Based Software Understanding through Graphics).

have been interested in more details of the example and how they could be integrated into the solution. The protocol of Figure 5-2c exemplifies this phenomenon. The obviously useful parts of the example were handled first, without asking questions; the less obvious features were never addressed, because programmers decided to stop with the first prototype. Giving the programmers a picture of the goal was already suggested as one way to encourage a minimum of work on a problem. In any case, not anticipating this prototyping is a problem with the setup of the observations and nothing intrinsic to the tool.

In summary, the initial round of observations provided

- a few specific types of questions and features the tool should support;
- a greater understanding and appreciation of the intended end users of the tool; and
- a greater understanding and appreciation for established patterns of work and implications for the acceptance and integration of the EXPLAINER tool.

5.3 Observations with the First System

Once a prototype of the EXPLAINER tool had been developed, it was tested in the context of a class project in an advanced artificial intelligence programming class. Observing programmers with this version of the system provided feedback on the current state and future direction of the implementation. The project consisted of the students studying the code for the TMYCIN system and answering twelve questions. The TMYCIN system, by Gordon Novak of the University of Texas at Austin, is a small expert systems shell based on the EMYCIN tool [Shortliffe 76]. At issue was how well the prototype EXPLAINER tool would support comprehension of a program example. As with the first round of observations, a second purpose was to get further experience that would aid in designing a formal experiment. The observations were carried out by a student, Mehran Majidi, for his masters thesis and additional information and background is reported in that thesis [Majidi 91; Majidi, Redmiles 91].

Table 5-1: Summary of Measurements Taken During the TMYCIN Observations

TMYCIN Observations		
	Users	**Nonusers**
Grade	22.83	17.00
Solution Time	122.13	157.00
Tool Time	82.84	n/a
Tool Time/Solution Time %	72.71	n/a

The context of a class was ideal for this informal evaluation of the prototype. First, it provided a homogeneous population of LISP programmers. Most were first or second year graduate students (Masters and Ph.D.) who had about the same amount of LISP experience. Second, the class was taught on a ''case'' basis. Students were learning about artificial intelligence programming by looking at scaled-down versions of major projects. The material was evolving into the book by Peter Norvig [1992] . Third, the study of the TMYCIN system was a usual component of the course and students thought the tool would be helpful. This last point makes the conditions of these observations more realistic: the students were using the tool basically for their own needs.

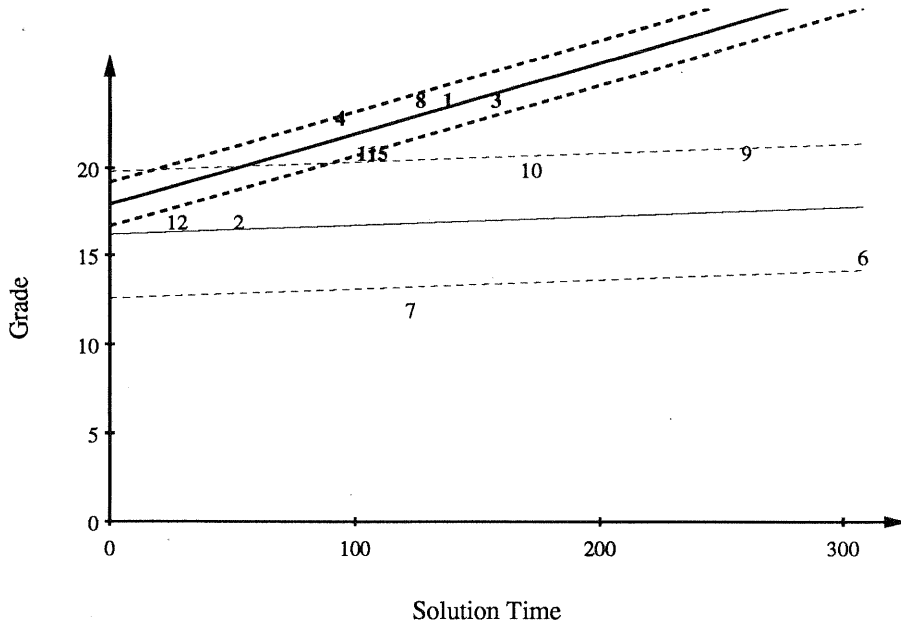
For studying the TMYCIN code, all students had a few pages describing how to call some major functions on a test data set. The students were free to run the system and look at the code listing. For comparison, half of the students worked under these conditions, while half had the option of supplementing their work by using the prototype tool. How much of the total project time was spent using the tool was recorded. Students were randomly assigned to one of the groups. In the end, both groups answered the same set of twelve questions scored at two points each. One typical question asked the student to provide a description of the function that performed backchaining. Students who used the prototype EXPLAINER tool gave comments during and after their work.

Figure 5-3 shows the appearance of the prototype tool. The interface was very similar to the final tool (Chapter 3), except there was no sample output pane and the text explanations were incorporated into the diagram nodes. The sample output pane had already been planned for the graphics domain. The separation of text from diagrams was part of an effort to make the text explanation more flexible and to make the diagram less cluttered. This modification was in part a result of comments made by users.

Table 5-1 gives the grade, total project time, time spent with the tool, and percentage of tool time out of total project time. A standard ANOVA analysis comparing the mean grades of the two groups yields $F(1,10)=15.75$ for a confidence of $p=.005$: the tool users scored better than the nonusers. For the solution times, no statistically significant difference in means could be detected. However, examining the measurements revealed that the slowest tool user finished faster than half of the nonusers—positive, if not statistically significant. Moreover, this finding also provides an explanation for lack of significance in the ANOVA test: the large overlap in the measured ranges.

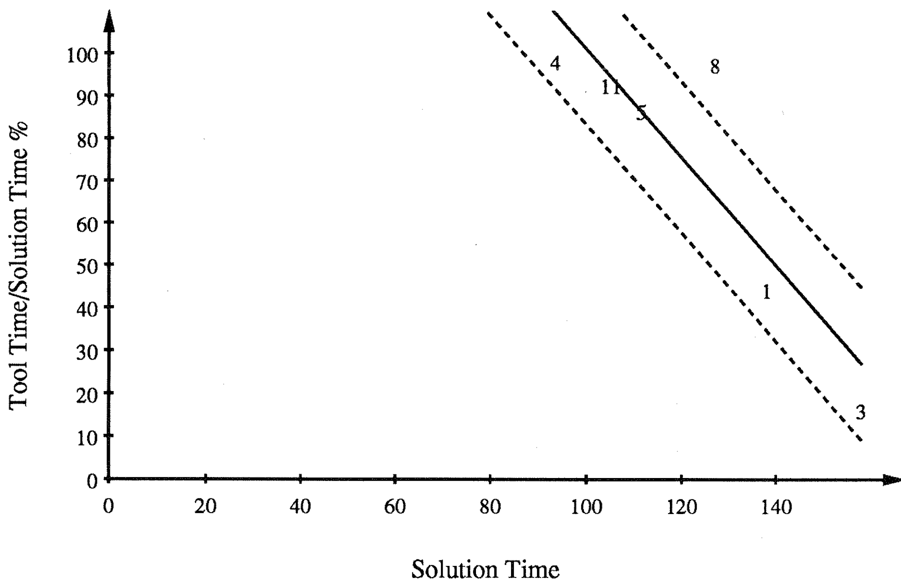
Two plots summarizing the results of the TMYCIN experiments reproduced by permission from Majidi and Redmiles [1991] are shown in Figure 5-4. Though the plots confound more than one observed variable, they illustrate some interesting trends. Figure 5-4a suggests that users of the explainer tool performed their question answering task better than nonusers (with respect to grade) and faster (with respect to time). Figure 5-4b suggests that the more use a user made of the tool, the faster that user finished the task. Both of these graphs and the trends they portray will be reexamined more carefully in Chapter 7.

The observations with TMYCIN emphasize the evaluation of EXPLAINER as a tool for comprehension, at least inasmuch as comprehension can be reflected in correct, descriptive answers to questions. Less subjective, and closer to the real goal of EXPLAINER, was that subjects were able to use the prototype tool to find the right points in the code to describe. Thus, even if the students were simply rewording text descriptions provided by EXPLAINER, their performance at least emphasizes the value of EXPLAINER's interface in helping users index specific system information. This result is especially meaningful given the size of the test example: TMYCIN consists of 49 function definitions and 6 variable definitions in about 570 lines of LISP code, about 100 of which are comment lines.



(a) Performance of Users and Nonusers

Users: Y = 17.93 + 0.04X, Standard Error = 1.26, Coefficient of Determination = 0.42, Coefficient of Correlation = 0.65. Nonusers: Y = 16.19 + 0.01X, Standard Error = 3.62, Coefficient of Determination = 0.03, Coefficient of Correlation = 0.18.



(b) Helpfulness of Tool (w.r.t. Time)

Y = 228.87 + -1.28X, Standard Error = 17.82, Coefficient of Determination = 0.78, Coefficient of Correlation = -0.88.

Figure 5-4: Trends in the TMYCIN Observations (reproduced by permission from [Majidi, Redmiles 91])

This correlations present least squares lines (solid) with standard error lines (dashed). (a) illustrates the trend that users (data shown in bold) performed better and faster than nonusers. (b) illustrates the trend that users finished faster the more use they made of the tool.

The students who used the prototype were also asked to comment on their experience. They indicated that the diagrams, the highlighting command and the text explanation were useful in answering the questions about TMYCIN. However, they wanted the highlighting to be more visible (underscoring was the method used for the prototype). The users recommended that the source code and the diagram should be presented in an editor environment. An editor environment would provide uses with the ability to vary some code parameters for experimentation. The users also felt that the editor environment would allow them to search for a node or a LISP function. This wish emphasizes a need to better integrate the EXPLAINER tool with the location research (Chapter 3), especially if larger examples are being used.

The insights gained from this study contributed in several ways to the design of the formal experiment described in the next chapter. In the formal testing, comprehension was given a more specific and practical definition. Subjects were evaluated both on how well they could find the appropriate points of an example code and apply them in the solution to a task. This definition reflects a greater emphasis on comprehension by requiring more cognitive effort on the part of the programmers than simple recapitulation of retrieved text.

Another adaptation for the formal test was that less work had to be demanded of each subject. The TMYCIN comprehension task required so much time that students had to schedule several sessions for working with the tool. For better control, and in order to test with a larger number of people, a smaller example and task would be chosen for the formal study.

In summary, the second round of observations provided

- a semi-realistic environment for problem solving and motivation;
- confirmation that the prototype implementation supported some aspects of comprehension;
- evaluation of the approach applied to a large task and example; and
- additional feedback for future design of the tool and of formal, user testing.

5.4 Summary

These initial, informal observations guided the development of the EXPLAINER implementation and its conceptual framework, as well as laying the foundation for formal testing of the tool as will be described in the next chapter. Specific contributions of the two rounds of observations have already been summarized. In Chapter 7, following the description of the formal experiment, related trends that appeared across the observations and formal experiment will be discussed. Specifically, the issue of percentage of time a tool is used will be reviewed.

6. Evaluation through a Formal Experiment

6.1 Goals

The general goals of this phase of evaluation were to verify the effectiveness of the example-based approach to programming, testing the EXPLAINER interface in particular. A controlled empirical study was performed with primary experimental goals of determining

- whether programmers using the EXPLAINER tool and its example-based approach would perform programming tasks ''better'' than programmers using alternative approaches; and
- what particular aspects of EXPLAINER would help programmers the most or the least.

Secondary goals were to observe

- how well EXPLAINER served as a framework for studying the use of examples; and
- in general, what use programmers made of examples.

6.2 Design

It was decided to compare the performance of programmers using EXPLAINER to the performance of programmers using the Symbolics DOCUMENTEXAMINER. The DOCUMENTEXAMINER tool is currently available as part of the normal Symbolics programming environment. Thus, it runs on the same system as EXPLAINER and has descriptions covering the same function library that EXPLAINER is intended to help programmers use. However, comparing EXPLAINER directly to the DOCUMENTEXAMINER is somewhat unfair and uncontrolled. In the end, it was decided to compare programmers' performance under three conditions using a between-subjects design. Table 6-1 summarizes the variations between these conditions and in addition, one possible extension. The reasoning behind the selections follows below.

First, the essential premise of the EXPLAINER tool and approach is that programmers work from an example similar to the task they are trying to solve. Thus, a direct comparison between the EXPLAINER tool and the DOCUMENTEXAMINER would leave open the question of whether it was the example that helped or the tool. This confusion was resolved by allowing subjects using the DOCUMENTEXAMINER (Group D) to have access to the same example which subjects using EXPLAINER had. Thus, both DOCUMENTEXAMINER and EXPLAINER subjects had the same starting point, just different tools with which to comprehend and develop that starting point (i.e., the example). This experimental design emphasizes the evaluation of EXPLAINER as a tool for comprehension.

A second problem is that the EXPLAINER tool embodies an approach to filtering and organizing information about software as well as an interface for presenting and accessing that information. Comparing EXPLAINER to the DOCUMENTEXAMINER, even supplementing the latter with an example, would still vary both the information made available and the critical component of the interface, the interaction menu. This variation was reduced by having one group of programmers use EXPLAINER but not have access to the interaction menu (Group ''O'' for menu off). This choice emphasizes the issue of the explanation-based approach and allows a finer differentiation between the contribution of the interface to EXPLAINER versus its information content. The group that uses EXPLAINER fully operational, with the interaction menu, is termed ''E'' for EXPLAINER.

Note that Group D still might vary from Group O in two ways: in terms of the presented information, which was intended, but also in terms of the interface. This latter variation is not considered significant since the DOCUMENTEXAMINER interface consists primarily of the command to retrieve a documentation topic and thereafter only scrolling is necessary. Thus, the critical aspect to the interfaces seen by the D and O groups was, in practice, equivalent: i.e., scrolling the information presented on the screen.

Ideally one more condition might have been tested, in which subjects used the DOCUMENTEXAMINER without the example. The size of the subject pool made this infeasible. Furthermore, it was expected that little practical information would be gained from evaluating such a group. The very reason for undertaking the EXPLAINER project was the evidence, both from experience and the literature as discussed

Table 6-1: Control Groups in the Clock Experiment

This table is best read top to bottom. Groups were designed to isolate variation among information and interface. The example remained constant in the three groups actually tested (E, O, D). A fourth group (N) would have tested the effect of providing no example, but this was deemed of little practical value (as explained in the text).

Condition	Information Source	Example	Interface
E	Explainer	Cyclic Group Example	Menu & Scrolling
O	"	"	Scrolling
D	Document Examiner	"	"
N	"	No Example	"

earlier, that programmers use and gain leverage from examples in the form of previously written code. While documentation incorporates some examples, the content, especially in the case of the DOCUMENTEXAMINER, is often significantly different than that of the example provided to the D, E, and O groups. Again the intent was to maintain all groups on equal footing except for the variables of interest: specifically, the tools that supported comprehension of examples.

6.3 Subjects

Subjects over all the groups were selected from first and second year graduate students in computer science. All subjects had to have programmed some in LISP and have knowledge about computer graphics. Subjects were not familiar beforehand with the functions needed to do graphics on the Symbolics. This selection criteria helped ensure roughly equal ability, knowledge, and experience. These subject characteristics were intended to approximate non-expert, software professionals. The requirement that subjects program in LISP limited the size of the subject pool. A total of 24 subjects were tested, 8 in each of the three conditions.

Two exceptions occurred with respect to subjects during the testing. One subject, Subject #3, was dropped because of an error in experimental procedure: she was not given the example code from the beginning. She was replaced by adding a subject. One subject did not complete any parts of the assigned task but since she did work with the tools for a significant amount of time, did have the same experimental training, and did have the same before-experiment background. Her data was kept as part of the normal population sample.

6.4 Materials and Procedure

The setup for the three conditions was basically the same. Subjects were randomly assigned to a condition. Subjects sat in front of two consoles positioned side-by-side. The console on the right contained an editor window and an evaluation window. The console on the left contained a window running one of the help tools (EXPLAINER, menu-off EXPLAINER, or DOCUMENTEXAMINER). For DOCUMENTEXAMINER subjects, the example was place in a second window on the right-hand console so subjects would have a fixed version to refer to; the original example was always presented by the EXPLAINER tool.

The decision to have two consoles was made in order to simplify access to the help tool. Since subjects would not be familiar beforehand with the operating environment of the Symbolics LISP-machine, the notion of switching between exposed and hidden windows might artificially prevent use of the tool. With the two consoles, subjects could still ignore the help tool, but that would be more a purposeful choice.

Subjects were trained for approximately 15 to 20 minutes on a sample problem: a program to plot the progress of wins of two people playing one-on-one basketball. The supplied example was a two-axis graph comparing the progress of wins of two people playing squash. The subjects were shown the sample problem description and the example. The example included a code listing, a picture of the example output, and whatever additional information a subject accessed through the help tool, e.g., a schematic of

the program example. Subjects were told that a program similar to example program would solve the problem. They were shown, and allowed to practice, how they could access help on the example. For instance, for the DOCUMENTEXAMINER, they were shown how to find a topic description for a function or find candidate topics based on their own keyword. For EXPLAINER, they were shown how to select a fragment in one of the EXPLAINER panes and get additional information. Subjects were already familiar with the editor.

Subjects were told they could ask questions during the experiment. The experiment was not intended to test learnability of the interfaces but their usefulness. If a subject forgot how to expand a screen item in EXPLAINER or retrieve a topic in the DOCUMENTEXAMINER, they could ask. Subjects were not asked to think aloud as they worked on the task.

Thus, subjects would edit code on the right and access help on the left. The editor window was filled initially with the code for the example. Subjects had the choice of starting from scratch but none chose to do so. Subjects modified parts of the example code to adapt it to the solution. When subjects wanted to try out an intermediate solution, they signaled the experimenter who loaded and ran their current version. Subjects decided when the solution met the problem description. The active session, beginning with the training, was recorded on video tape for subjects who gave permission.

6.5 Measurable Variables

Mapping the goals of the experiment onto measurable data was not straightforward as the goals are abstract, i.e., not in direct correspondence to physical phenomena. For instance, what is meant by the phrase ''better performance'' used in describing the primary goal? Or even, how are ''programming tasks'' defined? This section refines the top level experimental goals into an initial set of measurable variables.

6.5.1 Defining the Task and Example

In a real world setting, programmers would normally seek help on a programming task they had chosen, or one chosen by a superior as part of the programmer's job. Key aspects to either situation are that the problem solvers might be very familiar with the problem task as usual part of their job and also that they have an interest in completing it. To maintain these aspects in part, the clock task was chosen because subjects would be familiar with the domain requirements. In addition, they were allowed to ask the experimenter any questions about the task, just as they might be able to ask a superior on the job. The experiment was not about refining requirements but about programmers' behavior in mapping a task onto a program solution. The task was also chosen because it might be fun for the subjects, thereby providing a substitute for the normal motivations.

It also seemed necessary to design a task that would be a non-trivial exercise for the subjects in order to have a greater chance of observing programmers' behavior. However, subjects were volunteers and could not be expected to spend more than about 90 minutes of their time. Thus it was decided that only one programming task would be used. In the end, the median time to complete the clock task was 40.5 minutes with the additional 15 to 20 minutes training and practice time on a separate example.

A long-term and more realistic goal of EXPLAINER is that many examples might be useful in solving a single task; programmers should extract code and ideas from each and combine the reused parts with new code. However, attempting this in the experiment would have required both more time from the subjects and greater familiarity with the Symbolics programming environment. Multiple editor buffers would have to be manipulated, multiple instances of the EXPLAINER tool would have to be juggled, and more knowledge of LISP might be needed.

Thus, the notion of ''programming task'' became, for this experiment, modification of an example to create a solution to a related problem. For purposes of observation, the modifications necessary to adapt the example to the solution were intentionally specific and isolated. The task description is shown in Figure 6-1. The code listing of the example is given in in Appendix II as Figure II-1 and the specific points needed to be modified are annotated. These points are referred to as *alteration points*. They may be thought of as subtasks for completing the assigned programming task. Each alteration point cor-

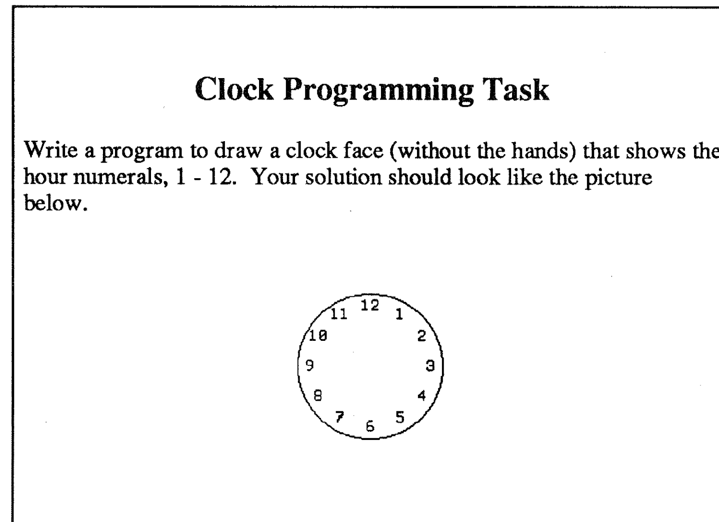


Figure 6-1: Description of the Clock Task as Given to Programmers

responds to a single subtask. For example, modifying Alteration Point B is where a programmer would add positions to make a total of 12 increments for the clock numeral labels. For purposes of illustration, Figure II-2 in Appendix II shows how Subject #23 modified these alteration points of the code example to solve the clock task.

6.5.2 Observing Subjects' Programming and Usage of Tools

Having designed the task and example as described above, observing programmer's behavior consisted of noting when different alteration points were modified. The time for each modification as well as any trial executions of intermediate versions of the program were recorded. This data supports an investigation of how subjects "converged" upon a solution: which alteration points were modified, which correct, when, and how many modification attempts were needed.

Actions within the help tool being used (either EXPLAINER or DOCUMENTEXAMINER) were monitored by recording keyboard and mouse actions. The only keyboard actions were the commands to the DOCUMENTEXAMINER to retrieve a documentation topic. The majority of actions in the DOCUMENTEXAMINER consisted of scrolling through a retrieved document. Actions in EXPLAINER with the menu off consisted solely of scrolling. Actions in EXPLAINER with the menu on consisted of retrieving and expanding information through the interaction menu and through clicking on fragments on the screen. Scrolling of retrieved information was used as well.

Scrolling actions provide a crude idea of where a subject's attention is focussed. For example, did the subject spend most of his or her time scrolling the code pane or the text information in EXPLAINER? These actions contribute to the larger picture of when a subject began and ended using the help tool. However, they can by themselves be misleading. The length of the information presented in the different panes of EXPLAINER varied: code listings took the greatest length while diagrams and the picture of the example program execution fit entirely in a pane. Thus, scrolling alone would be the worse indicator for a subject's use of the sample output pane in the EXPLAINER tool, whereas scrolling actions are the best and only indicator of subjects' use of the menu-off EXPLAINER tool. Scrolling actions were recorded as a list of four pieces of information: (i) time of action (ii) pane where action occurred (iii) identification of scrolling action and (iv) position scrolled to.

With the EXPLAINER system, mouse actions used to expand items on the screen and to activate menu actions were recorded as a list with five pieces of information: (i) time of action, (ii) pane where action occurred, (iii) identification of action, (iv) concept selected upon, and (v) perspective of the selected concept. This measure also is an indicator of subjects' attention and use of a tool. Though it is independent of the length of the information presented, it is not independent of the amount of information presented. Again, while the picture in the example output pane consisted of four fragments that could be selected, the listing in the code pane had hundreds.

Thus, taken together, the different types of keyboard and mouse data can provide an overview view of subjects' use of the help tools. The data provides a good indication of where subjects' were looking for information, and thereby can help not only in analysis of EXPLAINER but in planning where to put information in future interfaces.

6.5.3 Assessing Subjective Information

While the data described above provides an objective view of subjects' use of the tool, there are limitations to its use. For instance, in the EXPLAINER tool, clicking on the label "3" in the example output pane and using the "Highlight" command provides information equivalent to scrolling several times in the code pane and expanding a few items in the diagram pane. The objective enumerations of actions do not identify the importance of a program feature or piece of information. Some insight into these more subjective issues was gained by interviewing subjects after their session and collecting ratings and comments on specific aspects of the help tool used.

Subjects were asked to rate and comment on the usefulness of the information content and interfaces of the tools. For EXPLAINER, they were asked to individually rate the separate views: code, example output, diagram, and text. For the DOCUMENTEXAMINER, they were asked to rate the separate kinds of information in the documentation topics: syntactic summaries, textual descriptions, and the supplied example. When subjects were asked to give a rating, such as for the diagram pane, they were shown a scale with entries ranging, for example, from "not useful" to "okay" to "useful" to "very useful." They were allowed to select in-between two categories or even indicate "below the lowest." This data can be used informally to weight the raw keyboard and mouse data.

The interview also collected information about the subjects' satisfaction with particular aspects of the tool, as well as overall impressions and comments. This information can be used later to guide future improvements to the tool and evaluation procedure.

The interview is best described as a verbal survey; the questions followed a prepared form (see Figures II-3 and II-4 in Appendix II). It was believed that talking subjects through questions instead of having them fill out a form independently would permit more information to be elicited more accurately. The experimenter could begin with questions related to comments the subject may have made during the course of the test. Also, the experimenter could avoid gross inconsistencies by simply reminding the subject about earlier statments or actions that contradicted comments made during the post-experiment interview.

6.5.4 Defining "Better Performance"

The initial interpretation of "better performance" of programming tasks adhered to the following rough guidelines:

- better is faster;
- better is completing tasks and subtasks not otherwise solvable; and
- better is converging on a solution more directly, completing tasks with fewer trials and errors.

These guidelines are in accord with a belief that the EXPLAINER tool should help programmers get tasks done, not necessarily create solutions computer scientists would judge as general or elegant. Thus better performance is not defined to as producing a program that is extensible or has more parameters. The guidelines also support the belief that a tool should improve on what current methods allow. The point concerning convergence emphasizes a difference in the solution process and not simply an end result: i.e., it reflects how well programmers identify which software components support a solution to the current task and how.

Finally, how programmers feel about their experience using a tool may be subjective, but certainly affects their desire to use it. A tool that potentially could make a difference is of limited use if the intended users see no value in taking advantage of it or benefit from learning it. Thus

- better is also completing tasks with satisfaction and confidence in the tool.

The details of how ''better'' performance was calculated from the empirical data are given in Section 6.6. The experiment results showed some interesting patterns not anticipated, which suggested implications for scaling up the application of the EXPLAINER tool. In particular, the notion of convergence emerges as a critical consideration.

6.5.5 Glossary of Variables

The above subsections identified and described the kinds of variables that should be observed and how they would be related to higher level goals of the experiment. This subsection provides a glossary of specific terms used for these variables and may be used as a key to later tables and plots of the data.

Points Visited — the number of alteration points a subject modified;

Points Completed — the number of alteration points a subject changed correctly;

Changes — the total number of modifications a subject made while attempting to alter the example to produce a solution—these changes are counted as one per visit to an alteration point—while during the ''visit,'' several keystroke operations and entries may have been made;

Runs — the total number of runs or trial executions of intermediate versions of the solution;

Solution Time — the time in minutes that a subject took to complete the task —includes time spent using the help tool;

Tool Time — the amount of time the assigned help tool was in active use during the solution process.

In addition to this basic variables, Section 6.6.2 will introduce some derived variables. They are also summarized here.

Tool Time/Solution Time % — the percentage of tool time out of solution time, approximating how much usage a tool received during a session;

Changes/Points — changes per alteration points, averaging how many changes were made per correct alteration—factors out time and anomalies from incomplete solution and gives an indication of convergence or directness of solution;

Runs/Points — runs per alteration points averaging how many trial executions were used per correct alteration—supports evaluation of convergence;

Solution Time/Points — solution time per alteration points, averaging how much time was devoted to correctly modifying an alteration point;

ChangesB — changes per Alteration Point B.

The post-experiment interview collected subjective information identified by the following variables:

Overall — an overall rating of the tool based on satisfaction using it and consideration of its usefulness;

Interface — rating of just the interface used to access information presented by the tool;

Information — rating of just the information content presented by the tool.

In addition, each subject both ranked and rated the different information views provided by the tools. Variables for the ratings and ranks are identified by the view name: for EXPLAINER, Code, Picture (Sample Execution), Diagram, and Text; for DOCUMENTEXAMINER, Code (Supplied Example), Syntax (Syntactic Summary), and Text.

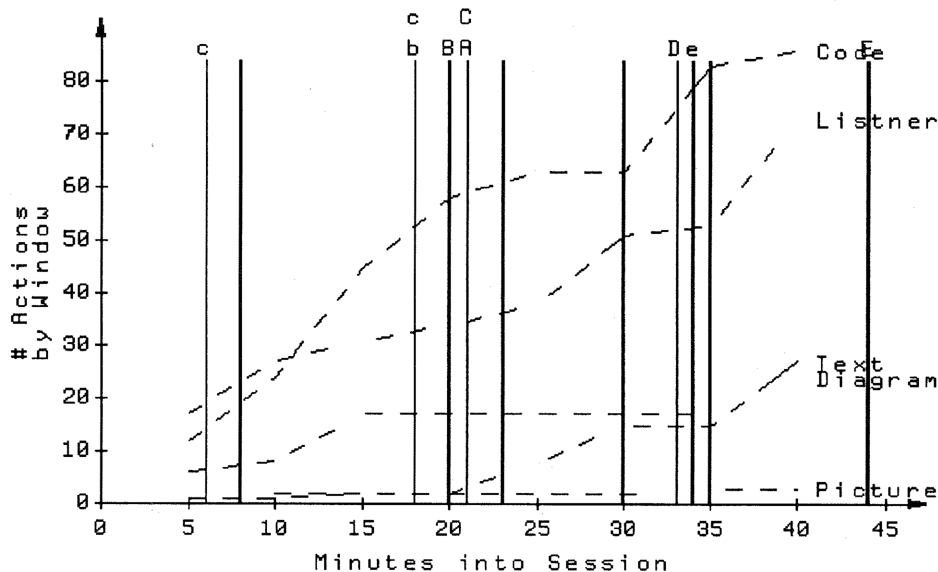


Figure 6-2: Actions in EXPLAINER Juxtaposed with Solution Progress

These figures represent encapsulated views of Subject #23's session. Subject #23 was in Group E—used EXPLAINER with the menu active. An explanation of how to interpret these graphs is given in Section 6.6.1

6.6 Results

6.6.1 Subjects' Solution Processes and Tool Usage

A total of about 7,000 items (about 70K Bytes) of keyboard and mouse actions were collected for the 24 subjects. This data can be simplified by applying different filters and then juxtaposed with the observations of the programming task activity. A view that was found especially useful resulted from plotting runs and changes made during the course of a session. Such plots reveal the nature of the design process followed by the subjects. Figures 6-2, 6-3, and 6-4 give these plots for a representative subject from each of the three groups. The three subjects whose sessions have been shown in the figures were chosen as the most typical representatives for their group based on their nearness to the medians for several critical variables: Points Visited and Complete, Changes, Runs, Solution and Tool Times. The use of the tool is also plotted according to the user interactions with the different views (Text, Picture or Sample Execution, Diagram, and Text) with the tool (EXPLAINER or DOCUMENTEXAMINER). These serve as an informal indicator of correlation between the tool and the subjects' design processes.

In the plots, the lighter vertical bars divide the time axis at times when changes were made to alteration points. The alteration point visited is labelled at the top of the bar. Lower case letters indicate points that were visited but not correctly altered. Capitalized labels indicate alteration points modified correctly. The darker vertical bars divide the time axis at times when an intermediate version of the solution was tried out by execution. Occasionally, runs and changes coincide within the minute precision of the measurements. The vertical axis corresponds to a count of subjects' interactions in a particular window. The dashed curves separate the counts according to which window the action occurred in. For convenience, the counts have been accumulated at 5 minute intervals.

The session overviews provide an initial, informal indication of the experiment results. Comparing subjects using EXPLAINER to menu-off EXPLAINER to the DOCUMENTEXAMINER, the design process required an increasing number of attempted changes to alteration points and of trial executions. Simultaneously, the use of the tool decreased.

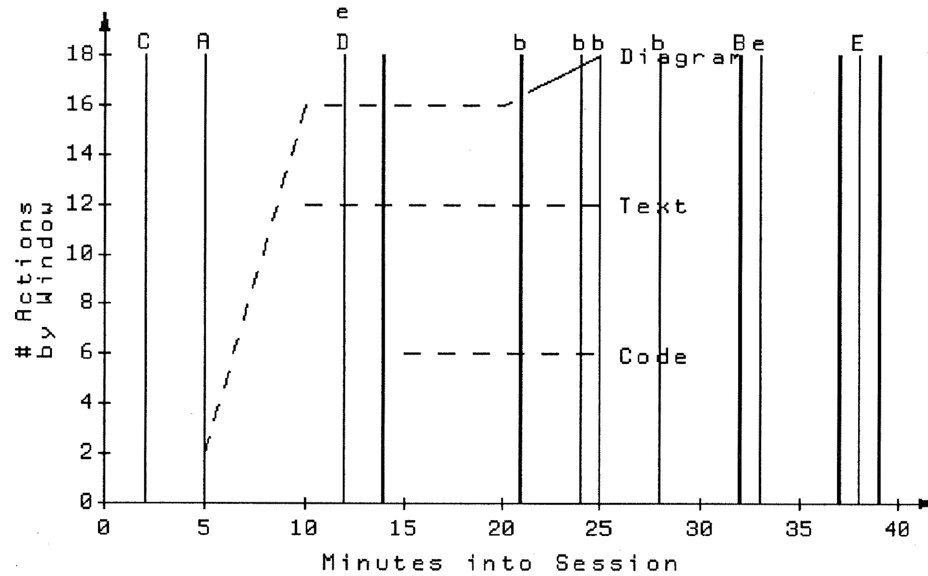


Figure 6-3: Actions in Menu-Off EXPLAINER Juxtaposed with Solution Progress

This figure represent an encapsulated view of Subject #4's session. Subject #4 was in Group O—used EXPLAINER with the menu off. An explanation of how to interpret these graphs is given in Section 6.6.1.

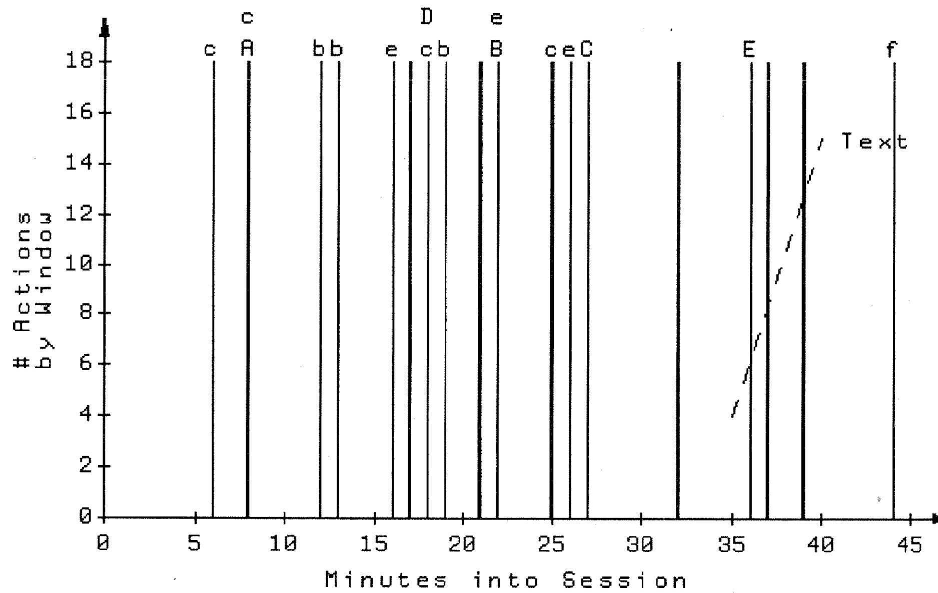


Figure 6-4: Actions in DOCUMENTEXAMINER Juxtaposed with Solution Progress

This figure represents an encapsulated view of Subject #20's session. Subject #20 was in Group D—used the DOCUMENTEXAMINER for help. An explanation of how to interpret these graphs is given in Section 6.6.1. The subject made use of the tool only late in the session: when attempting the hardest alteration points.

Table 6-2: Means of Primary Measures

	E	O	D
Points Visited	4.88	4.75	5.29
Points Completed	4.75	4.75	5.14
Solution Time	44.25	42.13	40.88
Changes	7.88	10.25	15.43
Runs	5.00	6.75	11.29
Tool Time	35.88	21.88	12.38

Table 6-3: Means Revised to Factor Out Point F
(See Section 6.6.2)

	E	O	D
Points Visited	4.63	4.63	4.71
Points Completed	4.63	4.63	4.71
Solution Time	43.25	39.00	34.88
Changes	7.38	9.75	13.14
Runs	4.75	6.50	9.57
Tool Time	35.50	19.00	11.50

6.6.2 Overall Quantitative Data

The quantitative data is summarized by means in Table 6-2. Corresponding to the proposed definition of ''better performance,'' the first variables checked in the basic data were the alteration points, both completed and visited, and the solution time. Unfortunately, Table 6-2 indicates very little difference in any of these variables and no significant location shift was detected by ANOVA. About the same alteration points were correctly modified in about the same amount of time. Thus EXPLAINER did not help programmers perform better with respect to solution time or quality as measured by number of correct alterations. However a significant difference in the amount of time tools were used was noted (for Tool Time, $F(2,21)=5.62$, $p=.05$).

Two anomalies in the alteration points information must be addressed. First, because it was deemed important that subjects made the decision when to stop, not all attempted the same number of alterations. This problem suggests that convergence be examined through ratios per actual alteration points correctly modified. For instance, instead of using the total count of the changes for completing the declared solution, the number of changes per alteration points would be used. Similarly, runs per alteration points and time per alteration points would be considered. This reasoning led to the addition of the ratio variables defined in the glossary of Section 6.5.5.

The second problem was that only 5 of 24 subjects correctly modified Alteration Point F and only 7 of 24 attempted it. It was particularly difficult and the graphical effect was not easily identified by looking at the terminal screen due to some slight distortion in the console. This problem suggests that measurements for subjects visiting Alteration Point F might be somewhat distorted. Namely, these subjects may have made disproportionately more changes or runs or taken more time simply because they were attempting an exceptionally difficult alteration. After reevaluating the data as originally collected, a way was found to subtract the effects of attempting F. (As seen in the Figures 6-2, 6-3, and 6-4 of the last section, times during which specific alteration points were visitied were maintained).

Table 6-4: Means of Calculated Ratios

	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
Tool Time/Solution Time %	78.54	54.43	33.47

Table 6-5: Means of Changes for Alteration Point B

	E	O	D
ChangesB	1.50	3.25	5.71

Table 6-3 summarizes the basic data revised to remove effects of F. The ratios described above are calculated with these revised measures and summarized in Table 6-4. Additionally, the percentage of tool time out of solution time is added. This variable was deemed useful with respect to the performance issue of how much a tool would be used and thus be able to affect the solution process.

Another solution to the problem of isolating effects from the more difficult Point F is to look at the changes made when changing individual alteration points. However, for the most part, it is not possible to make meaningful statistical comparisons among the individual counts because the range is often too small. Considered individually, only Alteration Point B (see Table 6-5) provides significant data ($F(2,20)=3.16, p=3.16$). It and the number of overall changes are what are used in the analysis below.

In sum, five calculated measures have been computed for the examination of performance: the ratios of changes per alteration points, runs per alteration points, and time per alteration points; the percentage of tool time out of solution time; and the number of changes in modifying Point B. The issue of performance can now be further explored without the anomalous effects of some subjects completing fewer than all predicted alterations, without variance due to the difficult Alteration Point F, and without the factor of time which was possibly affected by tool usage.

A final note before proceeding is that many of the standard statistical techniques cannot be relied upon in the comparisons that follow. With exception of time variables, ANOVA should not be used since many of the measures do not have a normal distribution. Furthermore, an argument will be made that the sample groups do not have the same variance. Though ANOVA is somewhat robust with respect to these assumptions (see Lindman [1974, pp. 31-34]), this robustness may reflect a lack of sensitivity to the data. In addition, many of the values are not continuous. In fact, comparisons of means are not always appropriate. In some variables, a value of zero or an entry of no value should be interpreted as infinitely bad. For example, after working with the problem for 26 minutes, Subject #12 had made no changes. A low number of changes is normally interpreted to be good, but no changes at all implies no alterations completed, and thus should be interpreted as bad.[3] These factors led to an investigation of methods based upon medians and ranks.

With these considerations as background, performance can now be examined in terms of the calculated measures. The focus is on characteristic differences in the solution processes of the different groups. In particular, the trends of tool usage, changes and trial executions, and variance are examined.

The results are presented for analysis in two tables, which compare medians, ranges, and 95-percent confidence intervals for basic measures (Table 6-6) and for the calculated ratios (Table 6-7). With this approach, hypotheses of location differences can be tested as follows. If a median from one sample group

---

[3]This subject was not discarded since the background was equivalent to other participants as noted in Section 6.3.

does not fall within the confidence interval around the median for another, it is concluded with the stated confidence that the median for the two samples differ. Trends can be seen by looking at how the absolute intervals shift. See Section 6.7.2.

With respect to the hypothesis that the EXPLAINER group proceeded in a more controlled manner than the DOCUMENTEXAMINER group, additional support could be given by demonstrating that the variances of the groups shifted higher, Group E to D. Confirmation of this trend is somewhat awkward. A T-distribution test exists [DuMouchel, Krantz 74, pp. 11-15 - 11-22] but assumes approximately normal distributions. The Squared Rank Test is distribution free but requires the use of means. As previously noted, calculation of means is complicated by the zero and null values from Subject #12. A reasonable way to proceed is to discard this subject's data for this one test. Results of the Squared-Rank Test as applied to the variables discussed in Section 6.7.2 are summarized in Table 6-8.

Table 6-6: Primary Measures: Medians, Ranges, and 95-Percent Confidence Intervals

	E	O	D
Tool Time	35.00 (16,64) (18,45)	21.00 (2,37) (8,29)	11.50 (0,34) (0,18)
Changes	8.00 (4,10) (5,9)	8.00 (3,23) (6,13)	12.00 (6.00,+INF) (6,27)
ChangesB	1.50 (1,2) (1,2)	3.00 (1,6) (1,5)	4.50 (1.00,+INF) (1,14)
Runs	4.00 (3,7) (4,7)	3.00 (3,19) (3,11)	7.50 (2.00,+INF) (4,28)

Table 6-7: Calculated Ratios: Medians, Ranges, and 95-Percent Confidence Intervals

	E	O	D
Tool Time/Solution Time %	86.24 (47.06,95.74) (64.29,91.43)	61.02 (4.35,94.87) (14.04,87.88)	21.13 (0.00,65.38) (2.44,61.54)
Changes/Points	1.60 (1.20,2.00) (1.25,1.80)	1.65 (1.20,4.60) (1.20,2.60)	2.40 (1.20,+INF) (1.20,9.00)
Runs/Points	0.90 (0.80,1.40) (0.80,1.40)	1.00 (0.60,3.80) (0.60,2.20)	1.50 (0.40,+INF) (0.80,9.33)

Table 6-8: Variances of Primary Measures and Calculated Ratios Compared by the Squared-Rank Variance Test

	E	D	|Z|	*p*
Tool Time	245.71	155.48	0.32	—
Tool Time/Solution Time %	277.19	533.38	0.66	—
Changes	4.55	67.14	2.91	.01
ChangesB	0.29	30.57	3.33	.01
Changes/Points	0.08	7.86	2.72	.01
Runs	2.21	78.29	2.91	.01
Runs/Points	0.08	9.71	2.70	.01

Table 6-9: Means of Measures from Interview

(a) Overall Ratings for Tools

	E	O	D
Overall	4.50	4.00	2.38
Interface	4.50	3.25	1.50
Information	4.25	4.75	2.00

(b) View *Ratings*

	E	O	D
Code	4.75	5.50	1.50
Picture	4.50	6.00	n/a
Diagram	3.75	2.63	n/a
Text	2.50	2.38	2.13
Syntax	n/a	n/a	1.00

(c) View *Rankings*

	E	O	D
Code	1.50	1.25	1.00
Picture	2.75	2.13	n/a
Diagram	2.50	3.50	n/a
Text	3.25	3.13	1.38
Syntax	n/a	n/a	2.25

6.6.3 Interview Data

The post-experiment interview had two purposes. The first was to evaluate the subjects' satisfaction with the tool they used. This aspect of evaluation is critical in determining whether a tool might be accepted for use. A second purpose was to seek information about specific components and features of the tools. As noted earlier, the fact that a component received heavy use cannot by itself imply that that component is important or useful. The interview, along with comments made by the subjects during the experiment, guides future improvements.

The survey data is summarized in Table 6-9. The data is ordinal but, for convenience, the rating categories were mapped onto integers between 0 and 6. If a subject did not use a component, they did not provide a rating. These null entries are generally interpreted as bad, lowest possible, on the grounds that all component were expected to contribute to the solution of the problem. They are mapped to 0.

Since the data is ordinal, the Kruskal-Wallis Test is used to determine shifts in means. Table 6-10 summarizes the results of the Kruskal-Wallis Test for the interview data. Since the values for the

Table 6-10: Two-way Comparison of General Ratings from Interview using Kruskal-Wallis Test

(a) Mean Shift—E to D

	χ^2 1 d.f.	p
Overall	4.94	.050
Interface	8.77	.005
Information	5.89	.025

(b) Mean Shift—O to D

	χ^2 1 d.f.	p
Overall	3.08	.100
Interface	5.36	.025
Information	7.08	.010

(c) Mean Shift—E to O

	χ^2 1 d.f.	p
Overall	0.72	—
Interface	4.16	.050
Information	0.67	—

EXPLAINER and menu-off EXPLAINER groups did not differ significantly except in one variable, the rating for the interface, a three-way comparison of Groups E, O, and D would have been of little value. Instead, the comparison is split into three two-way comparisons: E to D, O to D, and E to O. These two-way comparisons accentuate the contrast among the two groups of EXPLAINER users and the DOCUMENTEXAMINER and, for completeness, provide the comparison between the two EXPLAINER groups. Similarly, for the rankings and ratings of the different views of the EXPLAINER interface, Table 6-11 compares the ratings between the two EXPLAINER groups, giving significance values computed by the Kruskal-Wallis Test.

6.7 Discussion

6.7.1 Subjects' Solution Processes and Tool Usage

Figures 6-2, 6-3, and 6-4 are intended to provide a rough overview of a subject's session. In as much as they reflect typical sessions for the three conditions, they serve to illustrate and support some of the implications explored as statistical hypotheses in the next section. One prominent tendency that the figures illustrate, comparing Group E to O to D, is the increasing number and frequency of changes and trial executions subjects used to accomplish alterations, and thus, the overall solution. A hypothesis evaluated in the next section is whether EXPLAINER users (Groups E and O) followed a more direct path to a solution whereas DOCUMENTEXAMINER users relied more on trial and error and their own existing knowledge.

Table 6-11: Two-way Comparison of Ratings and Ranks between Full and Menu-Off EXPLAINER using Kruskal-Wallis Test

(a) Mean Shift—E to O

	χ^2 1 d.f.	p
Code View Rating	2.14	—
Picture View Rating	6.67	.010
Diagram View Rating	1.29	—
Text View Rating	0.26	—

(b) Mean Shift—E to O

	χ^2 1 d.f.	p
Code View Rank	0.02	—
Picture View Rank	1.95	—
Diagram View Rank	5.00	.050
Text View Rank	0.01	—

A second noticeable tendency apparent in the figures is that less use was made of the tool, comparing from Group E to O to D, to the point that Subject #20 did not rely on the DOCUMENTEXAMINER for help except for a small period of time towards the end of the session, when attempting to alter point F, the hardest of the alteration points (see Figure 6-4). EXPLAINER users, on the other hand, exhibit a constant use of the tool throughout most of the session (Figures 6-2 and 6-3). An hypothesis explored in the next chapter looks at what a correlation between usage of the tool and other factors might imply.

6.7.2 Overall Quantitative Data

The variables measuring usage of the tool (Tool Time Tool and Time/Solution Time %) are compared by confidence intervals in Tables 6-6 and 6-7. The medians for the DOCUMENTEXAMINER group fall outside of the intervals of the values for the EXPLAINER group, even at the full range of the measurements. The medians all decrease, with their intervals shifting downward, as the data is reviewed from Group E to O to D. Thus, the EXPLAINER tool was actively used a greater percent of the time during the solution process than was the DOCUMENTEXAMINER. That the median usage (see Table 6-7) for EXPLAINER was around 86% (compared to 21% for the DOCUMENTEXAMINER) implies that EXPLAINER was in use throughout most of the solution process.

The variables measuring changes (Changes, ChangesB, and Changes/Points) and the variables measuring runs (Runs, Runs/Points) are also compared in Tables 6-6 and 6-7. The medians for variables from the DOCUMENTEXAMINER group all fall on the high side of the intervals (both 95 and 100%) for the EXPLAINER group. The trend is confirmed that fewer changes and runs were being used by the EXPLAINER subjects compared to the DOCUMENTEXAMINER subjects. EXPLAINER subjects were proceeding more directly toward a solution in the sense that they converged upon a solution with fewer changes and trial executions. The DOCUMENTEXAMINER subjects were mostly exhibiting trial and error behavior.

Further, the overall time that tools were actively engaged suggests a problem with the initial conclusion concerning overall solution time. Specifically, the overall solution time includes time subjects diverted to

using the tool. Since EXPLAINER groups (E and O) spent more of the overall time using the tool, they were handicapped compared to the DOCUMENTEXAMINER group (D). The handicap is subtle. One could argue that if EXPLAINER were the better tool, using it would get the job done faster. On the other hand, if the subjects using EXPLAINER (in either form) had had more experience with the tool, they might have been able to use it more quickly. DOCUMENTEXAMINER subjects were not so greatly affected since they elected not to make as much use of that tool.

Interpreted in conjunction with the usage patterns of the two tools, the implication is that EXPLAINER subjects were not only using their tool more, but were actually benefiting from its use. In the same vein, the question whether subjects in different groups were working with more or less control led to the comparison of variances for the different observed and calculated measures using the Squared Rank Test (Table 6-8). It is concluded that there is a significant difference in variance among the EXPLAINER and DOCUMENTEXAMINER groups in all variables except those involving the tool usage: the difference occurs in the variables measuring the solution process.

Emphasis has been placed on examining the solution process by the EXPLAINER and DOCUMENTEXAMINER groups. The conclusions about the directness of the approach and the variance may be interpreted as very positive. While these conclusions do not imply that EXPLAINER subjects "understood" more about the software components they were using in the solution, they support the conclusion that EXPLAINER subjects were able to more directly recognize and apply the components. For larger problems, this advantage may in fact begin to exhibit itself in the time and quality measures, as suggested by the TMYCIN observations. Moreover, there are sometimes circumstances under which there is not as much freedom to change and run as often as a programmer might desire, or more specifically, as often as the DOCUMENTEXAMINER subjects exhibited. Managers might desire circumstances where average programmers could work more predictably. Finally, future work might test whether there is in fact improved comprehension from using EXPLAINER or simply whether the effects of recognition and application are maintained in time. Such information would be related to using EXPLAINER in tutoring or maintenance tasks.

6.7.3 Interview Data

In short, the trends shown in Table 6-10 are good news. The test confirms a location shift for all but the overall and information variables in the EXPLAINER to menu-off comparison: overall, subjects liked the EXPLAINER tool better than the DOCUMENTEXAMINER. More specifically, they found the interface and information content more useful for the task. In the comparison between the two EXPLAINER groups, the EXPLAINER tool was found to have the better interface. This is especially good since the interface subjects used with menu-off EXPLAINER was simply scrolling through previously expanded information. Also positive is that the information ratings between the two EXPLAINER groups show no significant difference; that implies that the EXPLAINER subjects were roughly as satisfied with the information they were able to retrieve by themselves compared to what was already laid out for the menu-off group by the experimenter. This observation again supports the usability of the EXPLAINER menu.

With respect to the full EXPLAINER interface, the capability that users most often commented positively about was highlighting. This feature was used frequently, as illustrated in Figure 6-2b. In contrast, one menu-off subject (#8) specifically commented on the fact that it would be nice to see "what code does what" and was annoyed when told, after the test, that this capability had been hidden for his assigned condition.

The *how* and *why* commands tended sometimes to frustrate subjects, and this frustration may have led to the poorer rating of the text pane. In particular, subjects expected more information to be stored about the example, including descriptions of the variables. More care needs to be taken in reviewing information stored for an example and more guidelines are needed for the future. Subjects were also confused about selecting different perspectives when soliciting how and why information. This could be due to insufficient training. However, one planned improvement to the program is to attempt to automatically select perspectives based on context or previous questioning [Moore 89].

The data rating and ranking the different views presented by the EXPLAINER interface (see Table 6-11) indicate that the different views were ordered in preference by the EXPLAINER subjects as code, diagram,

picture, and text. The order for the menu-off subjects was code, picture, followed by a tie between diagram and text. The Kruskal-Wallis Test shows that the diagram view received a significantly worse ranking in the menu-off group than in the full interface condition. Perhaps the purpose or interpretation of the diagrams is more obvious or easier given a menu to expand them with from the start. Another significant measure was the high ranking of the sample execution pane by the menu-off subjects. They consistently gave this view the highest rating and selected it second in their ranking order. Perhaps for this group, the picture compensated somewhat for their inability to highlight corresponding code, text, and diagram fragments, the picture helped subjects ascertain "what code does what."

In general, subjects seem to equate the code listing with the concept of example and see additional views of the example as being supporting documentation. In most cases, asking subjects about the value of the code led to the response that, "without it, I couldn't have done anything." The value of the example in an unfamiliar environment could not be disputed. Hence, the high ranking of the code view. While EXPLAINER is based on the philosophy that an example consists of all its many perspectives, this viewpoint is perhaps better suppressed with respect to designing an interface in terms more familier to programmers. This general perception of the code as example does lend credence to the use of code as the basis or micro-structure of the representation.

It is also interesting to note that the content rating for the DOCUMENTEXAMINER group was very close to that for the text pane in the EXPLAINER groups. This is not too surprising since the content of the DOCUMENTEXAMINER is presented primarily in a textual format. However, it is encouraging that there is some consistency in this subjective aspect of the experiment.

Finally, to check if performance in terms of the number of alterations correctly completed influenced a subject's opinion of the help tool used, a linear regression was calculated to determine if any correlation existed between the number of alteration points correctly modified and the rating of the tool. In other words, was it the case that subjects who got a lot done on the task felt satisfied and therefore rated the tool higher? Fortunately, in the case of subjects who used full EXPLAINER, no correlation existed (coefficient of correlation is 0.00): EXPLAINER subjects rated the tool unbiased by how well they completed the task. With the menu-off EXPLAINER group, a correlation is perceptible (coefficient of correlation is 0.23) and by comparison, the DOCUMENTEXAMINER group shows a high degree of correlation (coefficient of correlation is 0.49). Thus, DOCUMENTEXAMINER subjects had a tendency to rate the tool in proportion to how well they completed the task. The possibility that performance influenced the rating cannot be rejected. One implication is that the DOCUMENTEXAMINER had little impact in the subjects' valuations just as it seems to have provided little support for their task.

The post-experiment interview questions have provided information that observation alone could not. In particular, this data supports the interpretation that EXPLAINER had a positive impact on the solution process while the effect of the DOCUMENTEXAMINER seems dubious. The EXPLAINER tool being well received in this test situation encourages efforts to expand the work to more realistic problem settings. Some specific improvements were suggested and positive features identified. In all, the initial impression is positive for EXPLAINER.

6.8 Summary

From the observations and measurements taken during the experiment,

- EXPLAINER subjects and DOCUMENTEXAMINER subjects completed about the same alterations in about the same amount of time.
- However, EXPLAINER users made fewer changes and runs in working toward the complete solution.
- Moreover, EXPLAINER subjects performed the modifications more directly and with less intersubject variability than DOCUMENTEXAMINER subjects.

This directness in behavior might prove a more critical factor in larger tasks.

Additionally, from the post-experiment interview,

- The positive effects attributed to EXPLAINER were confirmed by subjects.

- Subjects rated the EXPLAINER tool highly and identified some specific features as useful.

General satisfaction is necessary if a tool is to get used and thus have a chance to be effective.

As a framework for observing programmers' use of examples,

- EXPLAINER allowed for unusually precise observations. This is evidenced by the ability even to filter out the effects of specific alteration points.

Further improvement can be achieved when the modification capability is integrated with the comprehension component (see Chapter 3).

The general value of examples, especially in an unfamiliar setting, was generally acclaimed by the participants. Ideally, the EXPLAINER tool eventually could be relied upon to the degree that programmers leave many systems details uninvestigated before starting a task, relying on the example to provide efficient explanation, on demand. This on-demand mode of operation should appeal especially to casual users of more exotic features.

7. Implications of Observations and Evaluation

7.1 Trends and the Larger Picture

The two observations and the formal experiment each provided individual views of the process of solving programming problems with examples. They also provided data on the use of the EXPLAINER for supporting the comprehension of examples. The discussion in this chapter reviews some aspects of the observations and the experiment, focussing on common themes. The goal is to provide a few general conclusions in addition to what the individual evaluations provided separately.

The conclusions in this chapter are developed from examining correlations in the data. Korin [Korin 77, pp. 117-118] provides a good description of how correlation analyses can be interpreted: considering the coefficient of correlation to be ''high'' or ''low'' is best done in the context of similar sets of data, as opposed to assuming some absolute scale. Accordingly, in this chapter, the values computed in the correlations are used mainly in the context of comparisons. The comparisons will usually be between groups: users of the EXPLAINER tool and and nonusers, as in the observations with TMYCIN, or between users of EXPLAINER and users of the DOCUMENTEXAMINER, as in the formal experiment. The values compared will usually be the slopes of the linear regression equation for comparing magnitudes of trends and the coefficient of correlation for comparing relative significance. For the formal experiment, only data with the ''F'' effort subtracted will be used (see Section 6.6.2). Also in this chapter, for simplicity, the only EXPLAINER group data considered from the formal experiment will be from the group that used the menu interface unless explicitly noted.

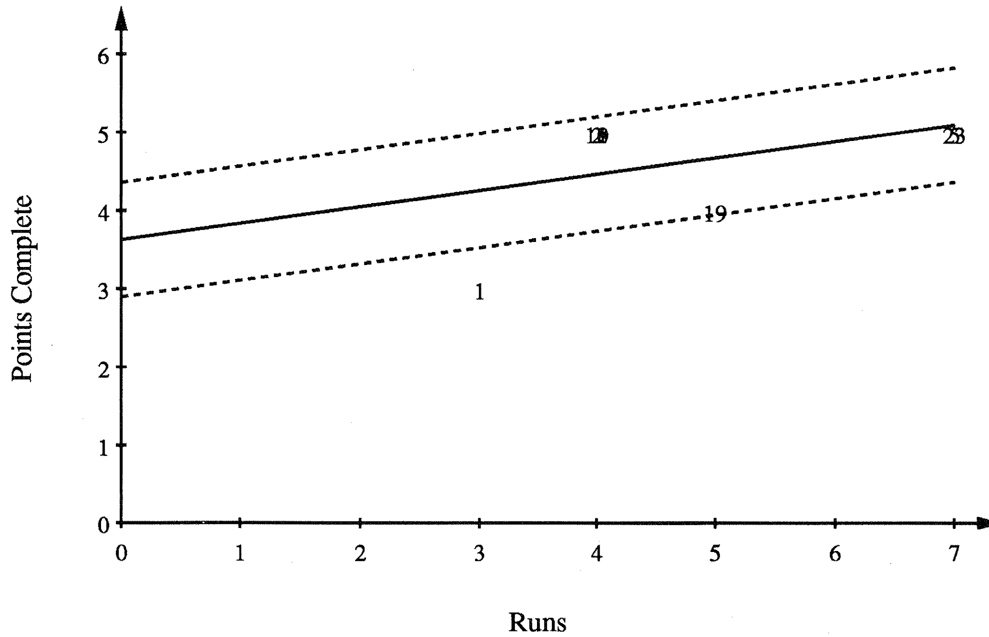
7.2 Prototyping Behavior

In the observations with the human consultant, one conclusion made was that a work pattern of prototyping might be affecting programmers' interest in asking questions and hence, might affect the possible effectiveness of the planned tool. It was also suggested that the EXPLAINER tool would be able to accommodate this approach. In particular, programmers could attempt a prototype, review the results of a trial execution, and return to the EXPLAINER tool and the example for more explanation to support further development of the prototype solution.

With the data collected in the formal experiment, it is possible to examine the effects of prototyping more carefully. For each programmer's data, as the number of trial executions increases, the number of correct alterations increases. Eventually, a programmer sees satisfactory output from a trial execution and stops working. Thus, one would expect that prototyping behavior would be evident in a correlation between the number of trial executions and the amount of work accomplished. In this case, work is being measured by the number of alteration points correctly modified.

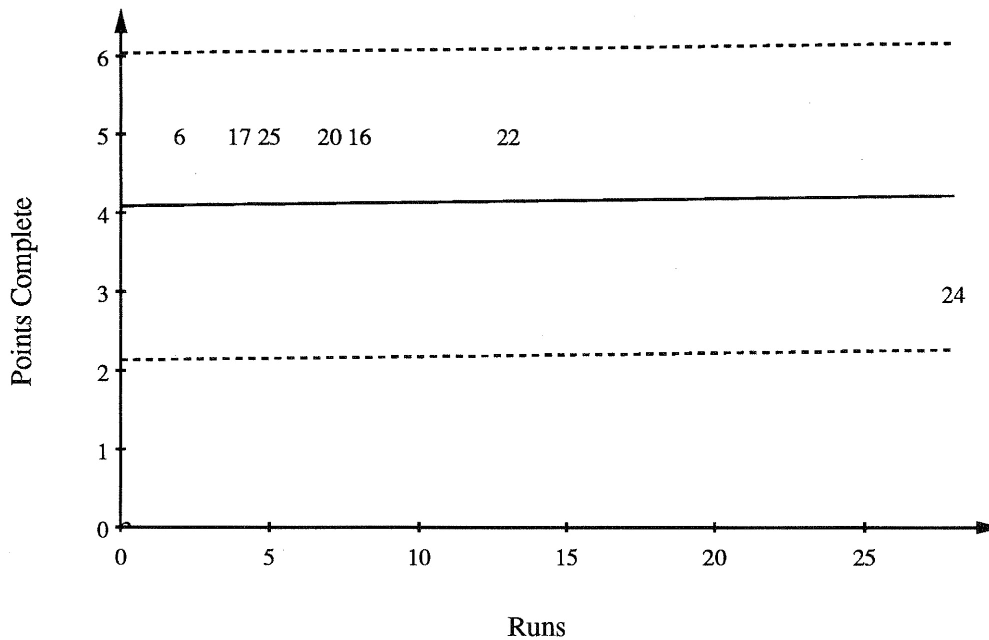
Figure 7-1 illustrates such a correlation between runs and correct alterations for both the EXPLAINER group and the DOCUMENTEXAMINER group from the formal experiment. The results are initially surprising. While subjects in both groups were prototyping, only the EXPLAINER group shows a high, positive correlation between prototype iterations (trial executions) and progress towards a solution (correct alteration points). (Compare the slopes and coefficient of correlation given in the figure.) In a sense, the trial executions in the EXPLAINER group would seem, from this correlation, to have had much greater and consistent impact on these subjects' success than had the trial executions on the DOCUMENTEXAMINER subjects.

Less formally, this same conclusion is implied by the difference in range of the run data for the two groups: 3 to 7 trial executions for EXPLAINER subjects and 2 to 28 for DOCUMENTEXAMINER subjects (excluding the subject who made no runs, #12). Refer also to Tables 6-6 and 6-7. Although the two groups were correctly completing about the same number of alteration points overall (Section 6.6.2), the runs were not as consistently effective for the DOCUMENTEXAMINER group. EXPLAINER emerges as being a tool supporting ''effective'' prototyping.



(a) EXPLAINER Users—Formal Experiment

Y = 3.63 + 0.21X, Standard Error = 0.73, Coefficient of Determination = 0.18, Coefficient of Correlation = 0.42.



(b) DOCUMENTEXAMINER Users—Formal Experiment

Y = 4.09 + 0.00X, Standard Error = 1.95, Coefficient of Determination = 0.00, Coefficient of Correlation = 0.02.

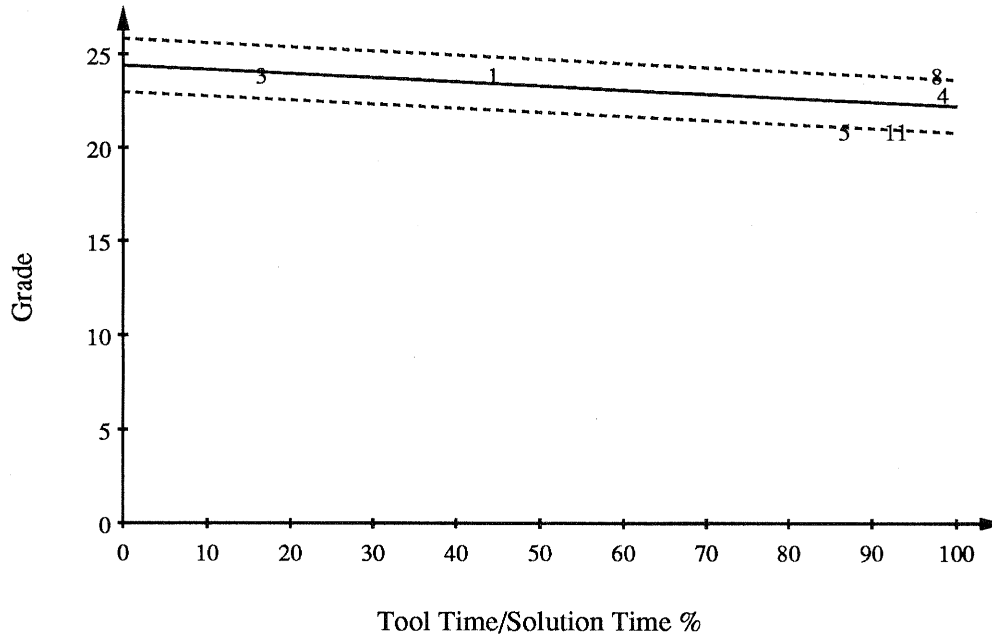
Figure 7-1: Complete Alterations vs. Runs—Effects of Tools on Prototyping

7.3 Effectiveness of Tools

The conclusion of the last section, and most of the conclusions about the formal experiment, deal with the effect of the EXPLAINER tool in leading to a more controlled problem-solving process. In the TMYCIN observations, this effectiveness was suggested by Figure 5-4b, which showed a trend that the time needed to complete the project of studying TMYCIN and answering questions decreased as tool use increased. "Increased use" was measured as the percentage of time the tool was in use out of the total time spent on the project.

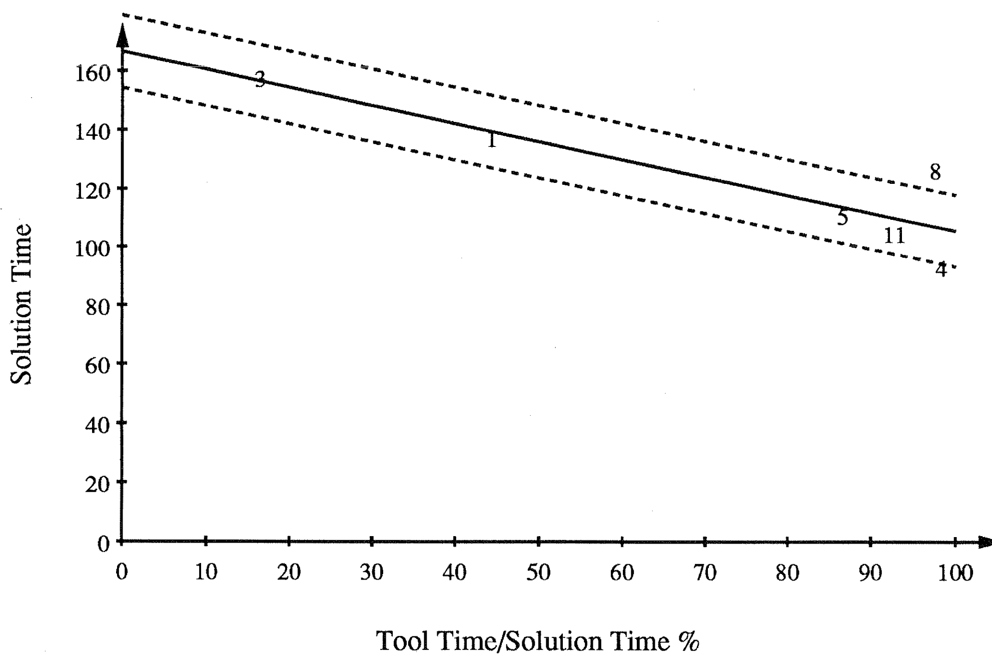
Separating out some variables reveals a finer grained, and more interesting, interpretation. To begin, although we observed a tendency that users making more use of the tool got done more quickly, they did not necessarily do better (though the EXPLAINER group did better overall). Figure 7-2a shows the correlation between grade and percent use of the tool. There is, in fact, a slight downward slope. Figure 7-2b illustrates the trend mentioned earlier, that the users who made more use of the tool got done faster (see also Figure 5-4b).

One possible interpretation is that the subjects who spent more of their time overall with the program needed more help to begin with and that the EXPLAINER tool helped "even out" their performance. Consider the correlation between the number of alteration points correctly modified and the percentage of tool usage for both EXPLAINER and DOCUMENTEXAMINER users in the formal experiment, Figure 7-3. Based on the sample data, the expectation is that programmers making greater use of the DOCUMENTEXAMINER, Figure 7-3b, will complete fewer alterations correctly; on the other hand, programmers making greater use of EXPLAINER, Figure 7-3a, are expected to complete slightly more alterations correctly. Assuming that programmers who rely more on the tool for help needed more help to begin with, then the conclusion is that EXPLAINER is going to make a positive difference for these people while the DOCUMENTEXAMINER is not. Thus, while in the previous chapter it was concluded that the EXPLAINER group (Group E) did not significantly perform better than the DOCUMENTEXAMINER group (Group D) with respect to number of correct modifications, there is a trend that EXPLAINER boosts the performance of programmers who would need more help. EXPLAINER is more effective than DOCUMENTEXAMINER for programmers needing more help.



(a) Grade vs. Percent Tool Usage

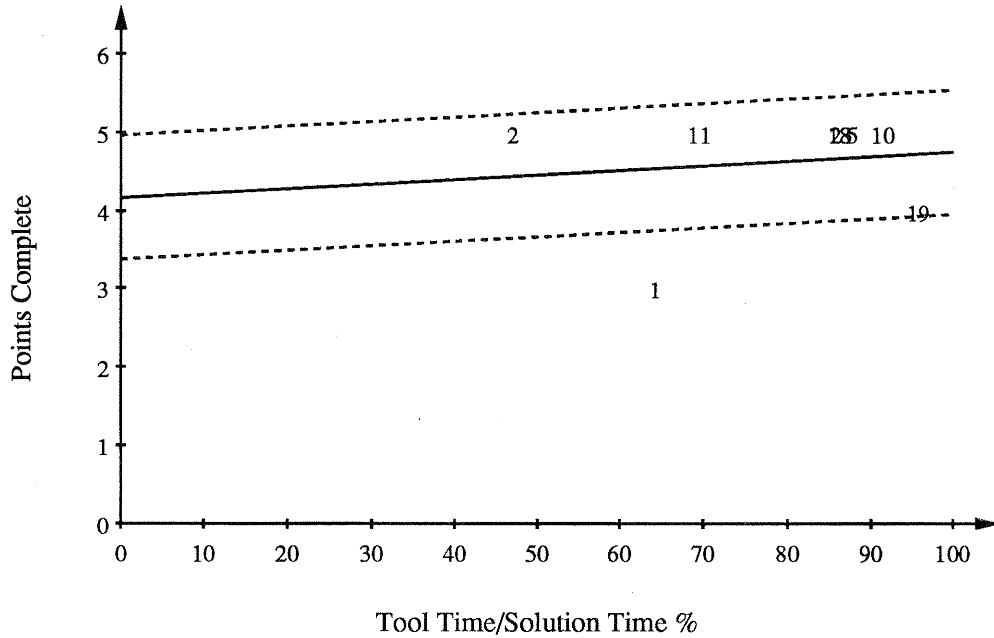
Y = 24.42 + -0.02X, Standard Error = 1.42, Coefficient of Determination = 0.26, Coefficient of Correlation = -0.51.



(b) Time vs. Percent Tool Usage

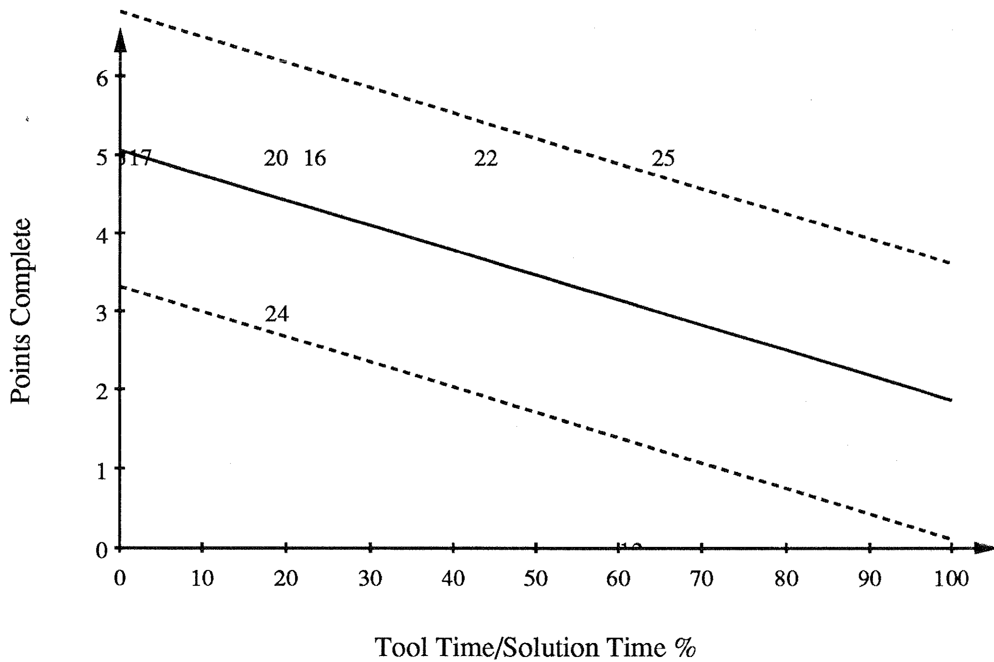
Y = 166.63 + -0.61X, Standard Error = 12.33, Coefficient of Determination = 0.78, Coefficient of Correlation = -0.88.

Figure 7-2: TMYCIN Observations—Effectiveness of Tool



(a) EXPLAINER Users—Complete Alterations vs. Percent Tool Usage

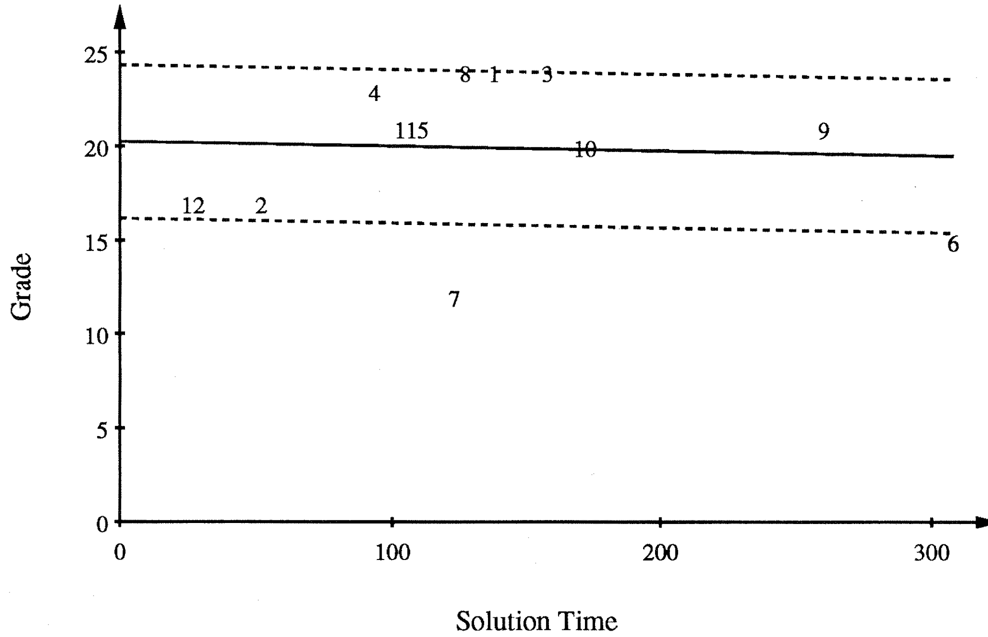
Y = 4.17 + 0.01X, Standard Error = 0.80, Coefficient of Determination = 0.02, Coefficient of Correlation = 0.13.



(b) DOCUMENTEXAMINER Users—Complete Alterations vs. Percent Tool Usage

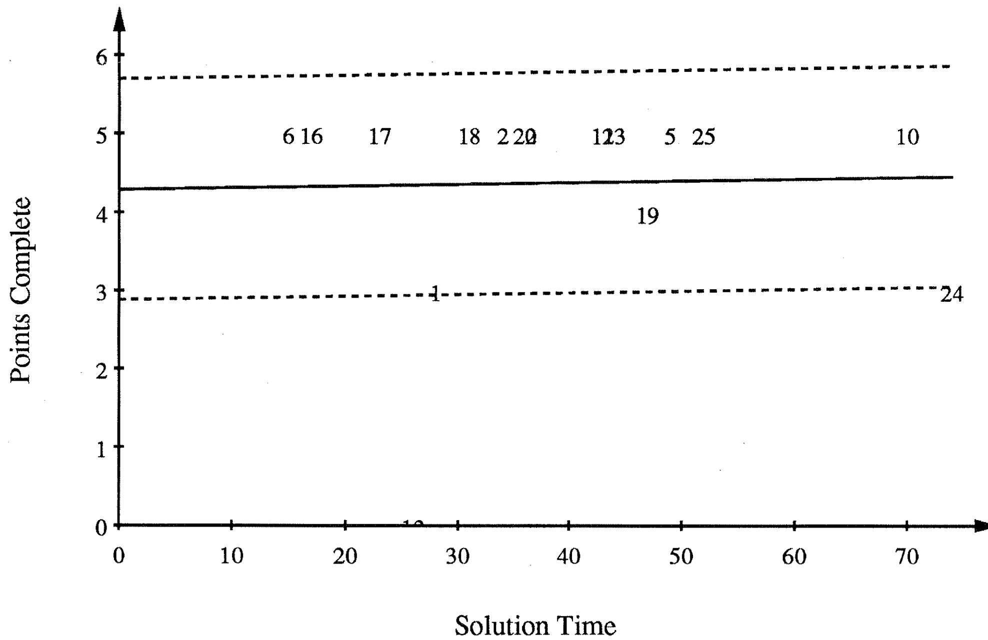
Y = 5.06 + -0.03X, Standard Error = 1.75, Coefficient of Determination = 0.20, Coefficient of Correlation = -0.44.

Figure 7-3: Formal Experiment—Effectiveness of Tool



(a) Both EXPLAINER Users and Nonusers—TMYCIN Observations

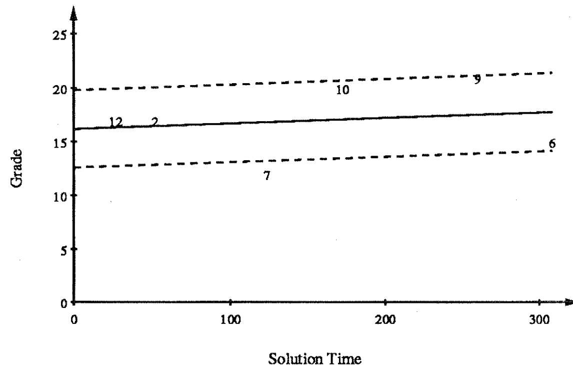
Y = 20.27 + -0.00X, Standard Error = 4.08, Coefficient of Determination = 0.00, Coefficient of Correlation = -0.05.



(b) Both EXPLAINER and DOCUMENTEXAMINER Users—Formal Experment

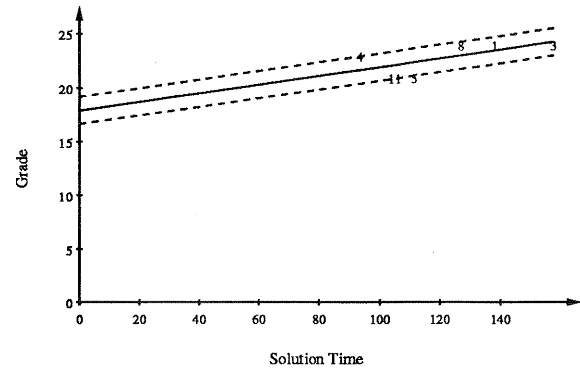
Y = 4.29 + 0.00X, Standard Error = 1.41, Coefficient of Determination = 0.00, Coefficient of Correlation = 0.03.

Figure 7-4: Work Accomplished vs. Solution Time



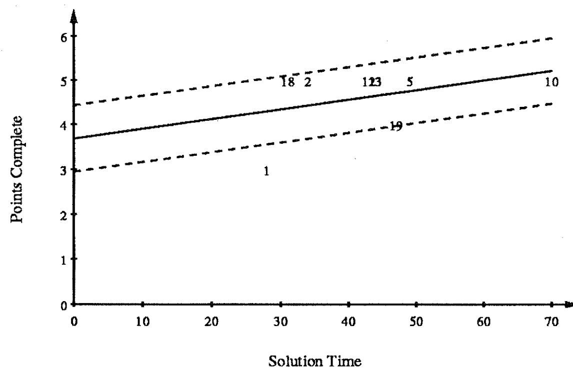
(a) EXPLAINER Users in
TMYCIN Observations

Y = 17.93 + 0.04X, Standard Error = 1.26, Coefficient of Determination = 0.42, Coefficient of Correlation = 0.65.



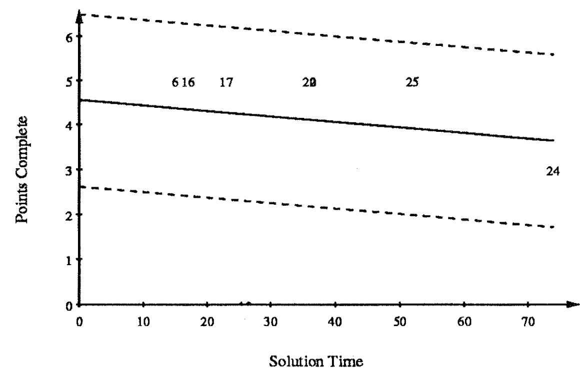
(b) Nonusers in
TMYCIN Observations

Y = 16.19 + 0.01X, Standard Error = 3.62, Coefficient of Determination = 0.03, Coefficient of Correlation = 0.18.



(c) EXPLAINER Users in
Formal Experiment

Y = 3.69 + 0.02X, Standard Error = 0.74, Coefficient of Determination = 0.15, Coefficient of Correlation = 0.39.



(d) DOCUMENTEXAMINER Users in
Formal Experiment

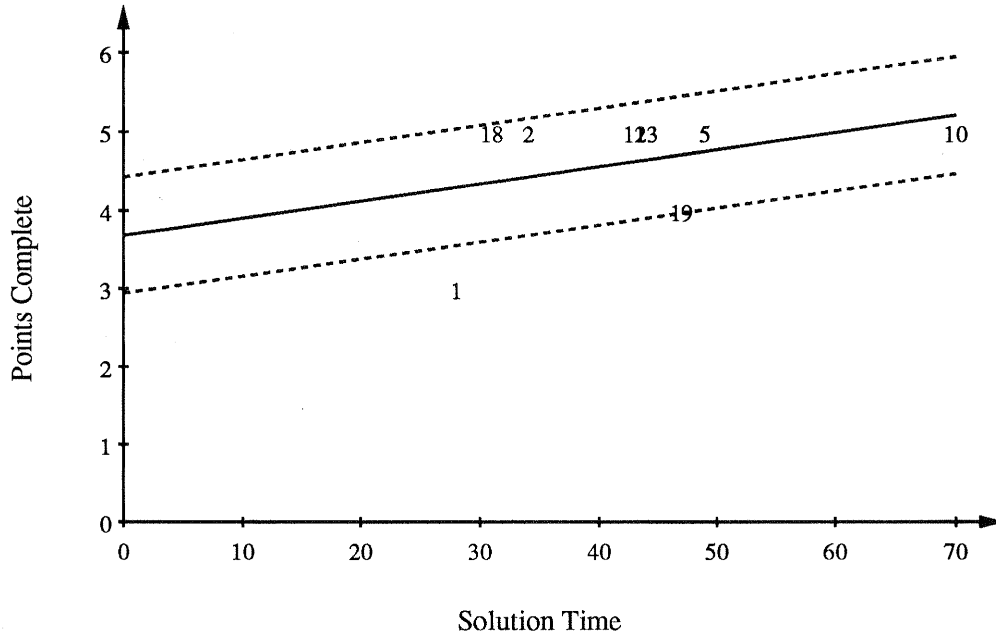
Y = 4.55 + -0.01X, Standard Error = 1.93, Coefficient of Determination = 0.02, Coefficient of Correlation = -0.13.

Figure 7-5: Solutions vs. Time According to Groups

7.4 Effort and Payoff in Problem Solving

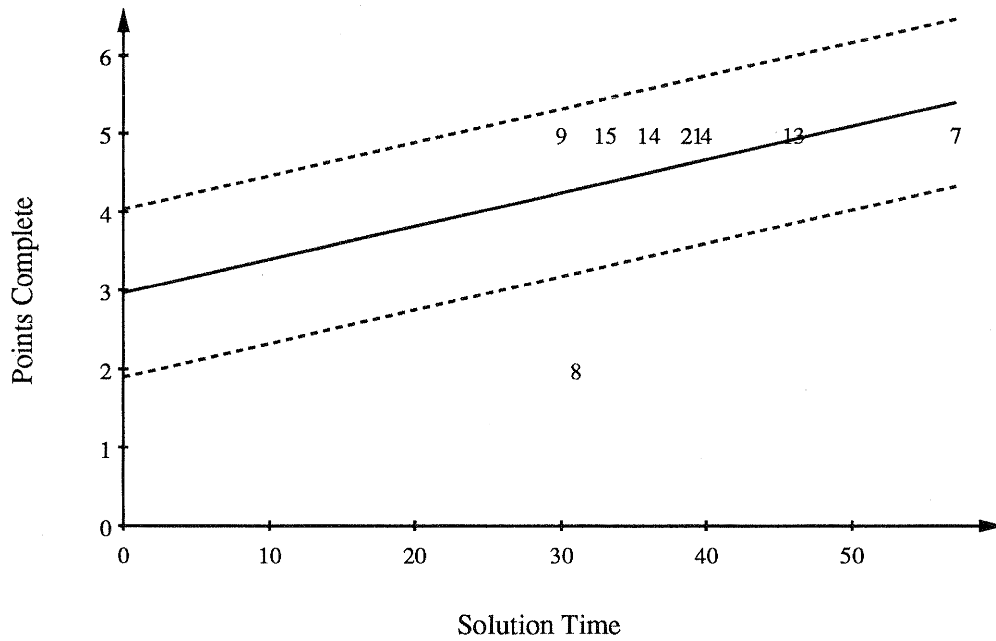
The previous discussion leads to the issue of the overall time.[4] Perhaps it was simply the case that subjects did better when they spent more time working on a problem. However, analysis of the overall data from both the TMYCIN observations and the formal experiment reveals no consistent correlation between total time spent and quality of solution (see Figure 7-4). Examining the data for each group separately shows that nonusers in the TMYCIN observations (Figure 7-5b) exhibited a slight improvement in grade with time invested, whereas users (Figure 7-5a) exhibited a much higher correlation and a slightly better improvement ratio. DOCUMENTEXAMINER users in the formal observations (Figure 7-5d) exhibited overall a decrease in alteration points correctly completed as the time increased, while EXPLAINER users (Figure 7-5c) exhibited a significant improvement and higher correlation.

[4]Looking at overall time also allows an examination of the effectiveness in a case previously overlooked: the control group who lacked a tool in the TMYCIN observations.



(a) Full EXPLAINER Users—Complete Alterations vs. Solution Times

Y = 3.69 + 0.02X, Standard Error = 0.74, Coefficient of Determination = 0.15, Coefficient of Correlation = 0.39.



(b) Menu-Off EXPLAINER Users—Complete Alterations vs. Solution Times

Y = 2.97 + 0.04X, Standard Error = 1.07, Coefficient of Determination = 0.13, Coefficient of Correlation = 0.36.

Figure 7-6: Start-up Time and Projected Correct Alterations

The graphs (Figure 7-5) also show that the minimal solution time for the groups using EXPLAINER was higher than that of the DOCUMENTEXAMINER users and the users without a tool. Using the tool certainly involves some start-up time and effort. A speculation is that this start-up period and subsequent use of the tool during problem-solving is helping the users. That is, the tool and effort it requires leads the users to become more involved with the problem-solving situation in a way that correlates overall with better solutions. Further support for this hypothesis is suggested by comparing the correlation between correct alterations and overall time between the full-EXPLAINER users and the menu-off users in the formal experiment (Figure 7-6). Menu-off EXPLAINER required less start-up time since users did not have to use a menu to uncover information; it was already expanded for them. The linear regression yields a lower y-intercept for the menu-off users than the full-EXPLAINER users. Thus the faster start-up is at the cost of the projected number of correct alterations.

The examination of overall time and grade per alteration points correctly modified supports additional conclusions. First, in the TMYCIN observations, working without a tool was less effective than working with the tool. Second, the overall trend is not that spending more time with a problem helps, but that spending more time with the EXPLAINER tool does. Third, a slow start-up is not necessarily bad, and the start-up period forced by the EXPLAINER tool may in the long run be of benefit to users.

7.5 Summary

In summary, the trends appearing across the observations and the formal experiment support the conclusions that EXPLAINER

- supports a more controlled or focussed solution process;
- fits into, not replaces, prototyping, a major work pattern common among programmers;
- supports its users in an effective manner compared to existing tools; and
- seems to provide a payoff for its additional start-up cost.

8. Other Related Work

8.1 Knowledge-Based Software Engineering

Several software reuse systems maintain representations of what was referred to as the problem- or task-domain knowledge. They use formal, automated techniques to produce new programs. The REQUIREMENT APPRENTICE [Reubenstein 90] supports designers in reusing requirements specifications, called *cliches*, a commonly occurring structure that exists in most engineering disciplines. It uses the cliches for supporting designers to frame a problem; however, the approach is based on the waterfall model and does not allow designers to intertwine problem specification and solution construction. DRACO [Neighbors 84] also stores such requirements specification, but it uses formal approaches to automatically derive designs from the specifications and does not reuse the specifications for supporting designers to frame a problem.

LASSIE [Devanbu et al. 91] and ARIES [Harris, Johnson 91] overlap with our approach in their use of knowledge bases for supporting software reuse. However, LASSIE focuses more on the knowledge base in terms of the structure and representation of artifacts, and less on the integration of access methods. ARIES focuses on how to build a knowledge base for reusable requirements, but does not focus on supporting designers to form problem requirements. Our approach complements these aspects by stressing an integrated environment for designers to support both the concurrent development of requirements specification and solution construction and the delivery of prestored objects related to the task at hand.

8.2 Domain Knowledge and Design Recovery

Domain analysis techniques [Prieto-Diaz, Arango 91] have commonality with our approach in that they support software reuse in domain-specific contexts. They focus on capturing deterministic knowledge about behaviors, characteristics, and procedures of a domain but do not include heuristics and probabilistic rules of inference. They require a well-defined domain with an analysis before reuse is possible. In our approach, such knowledge is gradually constructed through end-user modifiability as designers constantly use the design environments.

The DESIRE system assists in design recovery for reuse and maintenance of software [Biggerstaff 89]. The emphasis is on how much information can be recovered from existing codes. Our approach assumes a knowledge rich approach, relying on new examples input by original designers and evolution of the knowledge base supported by end-user modifiability. We also focus on evaluating the use and usefulness of different kinds of knowledge by designers.

8.3 Case-Based Reasoning

The work in case-based planning and reasoning [Kolodner 91; Ashley 90; Riesbeck, Schank 89; Hammond 89; Alterman 88] overlaps with EXPLAINER. "Examples" are very similar to "cases." However, some subtle differences exists. Examples in EXPLAINER are intended for human consumption whereas cases in case-based systems are intended primarily for automated reasoning; in EXPLAINER, representations for automated mutation have been foregone for representations for presentations. Cases, in the strict sense, are grounded in real experiences: how a specific, real subway system is organized [Alterman 88], some actual recipes [Hammond 89], or actual court cases [Ashley 90]. Examples can be real experiences or contrived situations. Example- and case-based systems both share the need to index examples. In EXPLAINER, indexes are based on concepts captured by different perspective networks. At a minimum, these indexes include graphic features and LISP programming constructs. Specification-linking rules in the CATALOGEXPLORER additionally index examples by specification needs. In addition to these types of indexes, case-based systems include indexes of failures [Hammond 89]. The possibility of negative examples has not yet been explored in EXPLAINER.

8.4 Summary

In general, the distinguishing principal in our approach centers around the role of human designers in the software development process. In particular, we stress the value of keeping designers closely involved in the development of an evolving software artifact by integrating domain-oriented knowledge into the location, comprehension, and modification cycle.

9. Conclusion

Experience based on the use of examples in software consulting led to research into the viability of an example-based tool for helping programmers use software components, such as functions in a software library. A model of design was developed that incorporated specification, location, comprehension, and construction. The model was designed to augment programmers, not replace them. A cooperative architecture allows components of the model to rely on the strengths of both the human and computer in solving programming tasks.

In general, the problem is analyzed as one of supporting programmers' development of an understanding of a task (situation model) and how a solution can be implemented on a computer (system model). The model augments programmers' problem solving through a problem-solving strategy of analogy. Exploring the overall structure of program examples (their macrostructure) and how that relates to programming language constructs (microstructure), helps programmers develop analogies and workable solutions (system model).

The model and implemented system components specifically help programmers with problems in using high-functionality computing systems:

- specification, and the ability to reformulate, helps programmers develop their goals and plans;
- specification and location help programmers learn about the existence of components and help them access examples;
- the examples help programmers see how to apply and combine components and what results are possible;
- construction supports the input of new examples and suggests ways programmers can be supported in modifying examples themselves.

Examples are represented and explained through multiple perspectives and views of these perspectives. The intent is to make it possible for programmers to recognize and identify aspects of the example program's macro- and microstructure for application to the solution of the current task. The visual domain simplifies the recognition of these structures as well as determines two minimally required perspectives: the graphics-features perspective and the LISP perspective. Users of the EXPLAINER tool may expand these and other perspectives and see the mapping between them.

The model has been successfully implemented and tested with several research and system building efforts. The testing has spanned the range from informal observations before implementation, to observations in a class setting, to formal ''laboratory'' comparisons.

Careful analysis and comparison of the results were both surprising and supportive of EXPLAINER: in an activity that normally exaggerates individual differences—programming [Egan 91]—the EXPLAINER users behaved as a group in a controlled and focussed fashion during problem solving. Also surprising was the evidence that the additional start-up cost of using the EXPLAINER tool correlated with a predicted payoff in terms of solution goals achieved; simply spending more time with a problem, even with the aid of other tools, did not help. EXPLAINER was found to be an effective tool, making a difference for programmers who relied upon it. EXPLAINER fits into an existing and common programming style—prototyping. Prototyping is in fact predicted by the needs of software engineering since users can not always have well-formed understandings of their problems or potential solutions on the computer.

The future goals include continuing critical evaluation of EXPLAINER and other systems supporting the example-based approach. Planned work includes researching how the approach and systems might be adapted in the object-oriented, software environment of a telephone company. Further work in supporting class projects is planned. The latter would provide long-term feedback about how students becoming accustomed to the EXPLAINER system work. It would reduce inexperience as a potential bias for evaluation. The example-based approach is flexible in its application. By varying the number and composition of catalog examples, the tool and approach are adaptable to use as a resource, reuse gazetteer, or tutoring environment

The approach might be tried in settings other than programming. Fischer and colleagues have gained insight by working with a variety of applications of design environments including architecture [Fischer,

McCall, Morch 89], user interface [Lemke 90], telephone-voice dialog [Sumner et al. 91], natural resource management [Lemke, Gance 91], lunar habitat [Stahl 92], and local area networks [Fischer et al. 92b], How examples and problem solving from examples in more general design domains can be investigated.

As these goals are pursued, a better idea for the kinds, amount, and cost of inputing knowledge supporting examples can be developed. The hope is that in a constructive environment, ''cut and paste'' operations would carry along the knowledge that interprets and annotates the example. The knowledge base constituted by the catalog would grow incrementally. The initial expense for constructing a ''seed'' [Fischer et al. 92b] might be high, but then so are current consulting and training costs for commercial software. One of the goals of this dissertation was to determine if a knowledge-intensive approach would even be effective in practice, an assumption taken for granted by many researchers artificial intelligence. For EXPLAINER, the evidence is affirmative.

References

[Adelson, Soloway 88]
B. Adelson, E. Soloway, *A Model of Software Design*, in M.T.H. Chi, R. Glaser, M.J. Farr (eds.), *Nature of Expertise*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, pp. 185-208, ch. 6.

[Alterman 88]
R. Alterman, *Adaptive Planning*, Cognitive Science, Vol. 12, No. 3, 1988, pp. 393-421.

[Apple 89]
Apple Computer, Inc., *Macintosh HyperCard User's Guide*, Cupertino, CA, 1989.

[Ashley 90]
K.D. Ashley, *Modeling Legal Argument*, The MIT Press, Cambridge, MA, 1990.

[Bateman, Paris 89]
J.A. Bateman, C.L. Paris, *Phrasing a Text in Terms the User Can Understand*, Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (Detroit, MI), Morgan Kaufmann Publishers, San Mateo, CA, August 1989.

[Bein, Smolenksy 88]
J. Bein, P. Smolenksy, *Application of the Interactive Activation Model to Document Retrieval*, International Workshop on Neural Networks and their Applications, Nimes, France, 1988, (also published as Technical Report CCU-CS-405-88, Dept. of Computer Science, Univ. of Colorado-Boulder).

[Biggerstaff 89]
T.J. Biggerstaff, *Design Recovery for Maintenance and Reuse*, Computer, July 1989, pp. 36-49.

[Biggerstaff, Perlis 89]
T.J. Biggerstaff, A.J. Perlis (eds.), *Software Reusability*, ACM Press and Addison-Wesley Publishing Company, Reaing, MA, 1989, (in two volumes).

[Bobrow et al. 88]
D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon, *Common Lisp Object System Specification*, Document 88-002R, X3J13, June 1988.

[Bobrow, Winograd 77]
D.G. Bobrow, T. Winograd, *An Overview of KRL, a Knowledge Representation Language*, Cognitive Science, Vol. 1, No. 1, 1977, pp. 3-46.

[Brooks 87]
F.P. Brooks Jr., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

[Chi et al. 89]
M.T.H. Chi, M. Bassok, M.W. Lewis, P. Reimann, R. Glaser, *Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems*, Cognitive Science, Vol. 13, No. 2, 1989, pp. 145-182.

[Clement, Gentner 91]
C.A. Clement, D. Gentner, *Systematicity as a Selection Constraint in Analogical Mapping*, Cognitive Science, Vol. 15, No. 1, 1991, pp. 89-132.

[Conklin, Begeman 88]
J. Conklin, M. Begeman, *gIBIS: A Hypertext Tool for Exploratory Policy Discussion*, Transactions of Office Information Systems, Vol. 6, No. 4, October 1988, pp. 303-331.

[Curtis 89]
B. Curtis, *Cognitive Issues in Reusing Software Artifacts*, in T.J. Biggerstaff, A.J. Perlis (eds.), *Software Reusability, Volume II: Applications and Experience*, Addison-Wesley Publishing Company, Reading, MA, 1989, ch. 13.

[Devanbu et al. 91]
P. Devanbu, R. Brachman, P.G. Selfridge, B.W. Ballard, *LaSSIE: A Knowledge-Based Software Information System*, Communications of the ACM, Vol. 34, No. 5, May 1991, pp. 34-49.

[Dijk, Kintsch 83]
T.A. van Dijk, W. Kintsch, *Strategies of Discourse Comprehension*, Academic Press, New York, 1983.

[DuMouchel, Krantz 74]
W.H. DuMouchel, D.H. Drantz, *Statistics Notes*, 1974, Course Notes Used at the University of Michigan.

[Egan 91]
D.E. Egan, *Individual Differences In Human-Computer Interaction*, in M. Helander (ed.), *Handbook of Human-Computer Interaction*, North-Holland, Amsterdam, 1991, pp. 543-568, ch. 24.

[Engelbart 88]
D.C. Engelbart, *A Conceptual Framework for the Augmentation of Man's Intellect*, in I. Greif (ed.), *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann Publishers, San Mateo, CA, 1988, pp. 35-66, ch. 2.

[Fairley 85]
R. Fairley, *Software Engineering Concepts*, McGraw-Hill Publishing Company, New York, 1985.

[Fischer 87a]
G. Fischer, *Cognitive View of Reuse and Redesign*, IEEE Software, Special Issue on Reusability, Vol. 4, No. 4, July 1987, pp. 60-72.

[Fischer 87b]
G. Fischer, *A Critic for LISP*, Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), J. McDermott (ed.), Morgan Kaufmann Publishers, Los Altos, CA, August 1987, pp. 177-184.

[Fischer 89]
G. Fischer, *Human-Computer Interaction Software: Lessons Learned, Challenges Ahead*, IEEE Software, Vol. 6, No. 1, January 1989, pp. 44-52.

[Fischer 90]
G. Fischer, *Communications Requirements for Cooperative Problem Solving Systems*, The International Journal of Information Systems (Special Issue on Knowledge Engineering), Vol. 15, No. 1, 1990, pp. 21-36.

[Fischer et al. 90]
G. Fischer, T. Mastaglio, B.N. Reeves, J. Rieman, *Minimalist Explanations in Knowledge-Based Systems*, Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr (ed.), IEEE Computer Society, 1990, pp. 309-317.

[Fischer et al. 91]
G. Fischer, A.C. Lemke, R. McCall, A. Morch, *Making Argumentation Serve Design*, Human Computer Interaction, Vol. 6, No. 3-4, 1991, pp. 393-419.

[Fischer et al. 92a]
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, Vol. 18, No. 6, 1992, pp. 511-522.

[Fischer et al. 92b]
G. Fischer, J. Grudin, A.C. Lemke, R. McCall, J. Ostwald, B.N. Reeves, F. Shipman, *Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments*, Human Computer Interaction, Special Issue on Computer Supported Cooperative Work, Vol. 7, No. 3, 1992, pp. 281-314.

[Fischer, Henninger, Redmiles 91]
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.

[Fischer, Lemke, Rathke 87]
G. Fischer, A.C. Lemke, C. Rathke, *From Design to Redesign*, Proceedings of the 9th International Conference on Software Engineering (Monterey, CA), IEEE Computer Society, Washington, D.C., March 1987, pp. 369-376.

[Fischer, McCall, Morch 89]
G. Fischer, R. McCall, A. Morch, *Design Environments for Constructive and Argumentative Design*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 269-275.

[Fischer, Nakakoji 91a]
G. Fischer, K. Nakakoji, *Empowering Designers with Integrated Design Environments*, in J. Gero (ed.), *Artificial Intelligence in Design'91*, Butterworth-Heinemann Ltd, Oxford, England, 1991, pp. 191-209.

[Fischer, Nakakoji 91b]
G. Fischer, K. Nakakoji, *Making Design Objects Relevant to the Task at Hand*, Proceedings of AAAI-91, Ninth National Conference on Artificial Intelligence, AAAI Press/The MIT Press, Cambridge, MA, 1991, pp. 67-73.

[Fischer, Nakakoji 92]
G. Fischer, K. Nakakoji, *Beyond the Macho Approach of Artificial Intelligence: Empower Human Designers - Do Not Replace Them*, Knowledge-Based Systems Journal, Vol. 5, No. 1, 1992, pp. 15-30.

[Fischer, Nieper 87]
G. Fischer, H. Nieper (eds.), *Personalized Intelligent Information Systems, Workshop Report (Breckenridge, CO)*, Institute of Cognitive Science, University of Colorado, Boulder, CO, Technical Report, No. 87-9, 1987.

[Fischer, Nieper-Lemke 89]
G. Fischer, H. Nieper-Lemke, *HELGON: Extending the Retrieval by Reformulation Paradigm*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 357-362.

[Fischer, Rathke 88]
G. Fischer, C. Rathke, *Knowledge-Based Spreadsheet Systems*, Proceedings of AAAI-88, Seventh National Conference on Artificial Intelligence (St. Paul, MN), Morgan Kaufmann Publishers, San Mateo, CA, August 1988, pp. 802-807.

[Fischer, Reeves 92]
G. Fischer, B.N. Reeves, *Beyond Intelligent Interfaces: Exploring, Analyzing and Creating Success Models of Cooperative Problem Solving*, Applied Intelligence, Special Issue Intelligent Interfaces, Vol. 1, 1992, pp. 311-332.

[Fisher 85]
G.L. Fisher, *Program Explanation Techniques*, Unpublished Ph.D. Dissertation, University of California, Irvine, Department of Information and Comptuer Science, 1985.

[Gentner 83]
D. Gentner, *Structure-Mapping: A Theoretical Framework for Analogy*, Cognitive Science, Vol. 7, 1983, pp. 155-170.

[Gick, Holyoak 80]
M.L. Gick, K.J. Holyoak, *Analogical Problem Solving*, Cognitive Psychology, Vol. 12, 1980, pp. 306-355.

[Girgensohn 92]
A. Girgensohn, *End-User Modifiability in Knowledge-Based Design Environments*, Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado, 1992, Also available as TechReport CU-CS-595-92.

[Goldberg 84]
A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley Publishing Company, Reading, MA, 1984.

[Green et al. 86]
C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich, *Report on a Knowledge-Based Software Assistant*, in C.H. Rich, R. Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Los Altos, CA, 1986, pp. 377-428.

[Hammond 89]
K.J. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*, Academic Press, 1989.

[Harris, Johnson 91]
D.R. Harris, W.L. Johnson, *Sharing and Reuse of Requirements Knowledge*, Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY), Rome Laboratory, New York, September 1991, pp. 65-77.

[Henninger 91]
S. Henninger, *Retrieving Software Objects in an Example-Based Programming Environment*, Proceedings SIGIR '91, Chicago, IL, October 1991, pp. 251-260.

[IEEE Software 87]
IEEE Software, July 1987.

[ISSCO 81]
ISSCO (Integrated Software Systems Corporation), *The DISSPLA User's Guide*, San Diego, CA, 1981.

[Jensen, Wirth 74]
K. Jensen, N. Wirth, *PASCAL: User Manual and Report, 2nd edition*, Springer-Verlag, New York, 1974.

[Keene 89]
S.E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley Publishing Company, 1989.

[Kernighan, Ritchie 78]
B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kintsch, Dijk 78]
W. Kintsch T.A. van Dijk, *Toward a Model of Text Comprehension and Production*, Psychological Review, Vol. 85, 1978, pp. 363-394.

[Kintsch, Greeno 85]
W. Kintsch, J.G. Greeno, *Understanding and Solving Word Arithmetic Problems*, Psychological Review, Vol. 92, 1985, pp. 109-129.

[Kolodner 91]
J.L. Kolodner, *Case-Based Reasoning*, 1991, in preparation.

[Korin 77]
B.P. Korin, *Introduction to Statistical Methods*, Winthrop Publishers, Inc., Cambridge, 1977.

[Krasner, Pope 88]
G.E. Krasner, S.T. Pope, *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Technical Report, ParcPlace Systems, Palo Alto, CA, January 1988.

[Kunz, Rittel 70]
W. Kunz, H.W.J. Rittel, *Issues as Elements of Information Systems*, Working Paper 131, Center for Planning and Development Research, University of California, Berkeley, CA, 1970.

[Larkin 89]
J.H. Larkin, *Display-Based Problem Solving*, in D. Klahr, K. Kotovsky (eds.), *Complex Information Processing: The Impact of Herbert Simon*, Lawrence Erlbaum Associates, Hilldale, NJ, 1989, pp. 319-341, ch. 12.

[Larkin, Simon 87]
J.H. Larkin, H.A. Simon, *Why a Diagram Is (Sometimes) Worth Ten Thousand Words*, Cognitive Science, Vol. 11, No. 1, 1987, pp. 65-99.

[Lemke 89]
A.C. Lemke, *Design Environments for High-Functionality Computer Systems*, Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado, July 1989.

[Lemke 90]
A.C. Lemke, *Knowledge-Based Framer Interface Generator*, IEEE Software, May 1990, pp. 35.

[Lemke, Gance 91]
A.C. Lemke, S. Gance, *End-User Modifiability in a Water Management Application*, Technical Report CU-CS-541-91, Department of Computer Science, University of Colorado, 1991.

[Lewis 87]
C.H. Lewis, *NoPumpG, EXPL, and Spatial Thought Dumper*, in G. Fischer, H. Nieper (eds.), *Personalized Intelligent Information Systems, Workshop Report (Breckenridge, CO)*, Institute of Cognitive Science, University of Colorado, Boulder, CO, Technical Report No. 87-9, 1987, ch. 13.

[Lewis 88a]
C. Lewis, *Some Learnability Results for Analogical Generalization*, Technical Report CS-CU-384-88, Department of Computer Science, University of Colorado, Boulder, CO, January 1988.

[Lewis 88b]
C. Lewis, *Why and how to learn why: analysis-based generalization of procedures*, Cognitive Science, Vol. 12, No. 2, 1988, pp. 211-256.

[Lewis et al. 90]
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.

[Lindman 74]
H.R. Lindman, *Analysis of Variance in Complex Experimental Designs*, W.H. Freeman and Company, San Francisco, 1974.

[Majidi 91]
M. Majidi, *Software System Understanding through Knowledge Based System*, Unpublished Master's Thesis, University of Colorado, Boulder, May 1991.

[Majidi, Redmiles 91]
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.

[Mann 85]
W.C. Mann, *An Introduction to the Nigel Text Generation Grammar*, in J.D. Benson, R.O. Freedle, W.S. Greaves (eds.), *Systemic Perspectives on Discourse: Selected Theoretical Papers from the 9th International Systemic Workshop*, Ablex, 1985, pp. 84-95, ch. 4.

[McIlroy 76]
M.D. McIlroy, *Mass Produced Software Components*, in P. Naur, B. Randell, J.N. Buxton (eds.), *Software Engineering*, Petrocelli/Charter, New York, 1976, pp. 88-98.

[Meteer et al. 87]
M.W. Meteer, D.D. McDonald, S.D. Anderson, D. Forster, L.S. Gay, A.K. Huettner, P. Sibun, *Mumble-86: Design and Implementation*, Technical Report 87-87, Computer and Information Science, University of Massachusetts, Amherst, MA, September 1987.

[Minsky 75]
M. Minsky, *A Framework for Representing Knowledge*, in P.H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill Book Company, New York, 1975, pp. 211-277.

[Moore 87]
J. Moore, *Explanations in Expert Systems*, Technical Report, USC/Information Sciences Institute, 9 December 1987.

[Moore 89]
J. Moore, *Responding to 'HUH': Answering Vaugely Articulated Follow-up Questions*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 91-96.

[Moore, Newell 74]
J. Moore, A. Newell, *How can MERLIN Understand?*, in L.W. Gregg (ed.), *Knowledge and Cognition*, Erlbaum, Potomac, 1974, pp. 201-252.

[Morch 88]
A.I. Morch, *CRACK: A Critiquing Approach to Cooperative Kitchen Design*, Unpublished Master's Thesis, University of Colorado, Boulder, May 1988.

[Mozer 84]
M.C. Mozer, *Inductive Information Retrieval Using Parallel Distributed Computation*, ICS Report 8406, Institute for Cognitive Science, University of California, San Diego, La Jolla, CA, June 1984.

[Myers 86]
B.A. Myers, *Visual Programming, Programming by Example, and Program Visualization: A Taxonomy*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 59-66.

[NBS 84]
NBS, *Guide to Available Mathematical Software*, Technical Report NBSIR84-2824, National Bureau of Standards, Gaithersburg, MD, January 1984.

[Neighbors 84]
J.M. Neighbors, *The Draco Approach to Constructing Software from Reusable Components*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 564-574.

[Norvig 92]
P. Norvig (eds.), *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Papert 80]
S. Papert, *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, New York, 1980.

[Pennington 87]
N. Pennington, *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Cognitive Psychology, Vol. 19, 1987, pp. 295-341.

[Pirolli, Anderson 85]
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.

[Prieto-Diaz, Arango 91]
R. Prieto-Diaz, G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamos, CA, 1991.

[Quillian 85]
M.R. Quillian, *Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities*, in R.J. Brachman, H.J. Levesque (eds.), *Readings in Knowledge Representation*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985, pp. 98-118.

[Rathke 86a]
C. Rathke, *The Browser: An Exploration Tool for ObjTalk Inheritance Structures*, Technical Report CU-CS-331-86, Department of Computer Science, University of Colorado, Boulder, CO, May 1986.

[Rathke 86b]
C. Rathke, *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache*, Unpublished Ph.D. Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

[Reeves 91]
B.N. Reeves, *Locating the Right Object in a Large Hardware Store -- An Empirical Study of Cooperative Problem Solving among Humans*, Technical Report CU-CS-523-91, Department of Computer Science, University of Colorado, Boulder, CO, 1991.

[Reiss 85]
S.P. Reiss, *PECAN: Program Development Systems that Support Multiple Views*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985, pp. 276-285.

[Reubenstein 90]
H.B. Reubenstein, *Automated Acquisition of Evolving Informal Descriptions*, AI-TR 1205, MIT, 1990.

[Rich, Waters 88]
C.H. Rich, R.C. Waters, *Automatic Programming: Myths and Prospects*, Computer, Vol. 21, No. 8, August 1988, pp. 40-51.

[Rich, Waters 90]
C. Rich, R.C. Waters (eds.), *The Programmer's Apprentice*, Addison-Wesley Publishing Company, Reading, MA, 1990.

[Riekert 86]
W.-F. Riekert, *Werkzeuge und Systeme zur Unterstuetzung des Erwerbs und der objektorientierten Modellierung von Wissen*, Unpublished Ph.D. Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

[Riekert 87]
W.-F. Riekert, *The ZOO Metasystem: A Direct-Manipulation Interface to Object-Oriented Knowledge Bases*, ECOOP'87, European Conference on Object-Oriented Programming (Paris, France), Springer-Verlag, Berlin - Heidelberg - New York, June 1987, pp. 132-139.

[Riesbeck, Schank 89]
C.K. Riesbeck, R.C. Schank, *Inside Case-Based Reasoning*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[Rist 89]
R.S. Rist, *Schema Creation in Programming*, Cognitive Science, Vol. 13, 1989, pp. 389-414.

[Schoen 83]
D.A. Schoen, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.

[Shipman, Chaney, Gorry 89]
F. Shipman, R. Chaney, T. Gorry, *Distributed Hypertext for Collaborative Research: The Virtual Notebook System*, Proceedings of Hypertext'89 (Pittsburgh, PA), ACM, New York, November 1989, pp. 129-135.

[Shortliffe 76]
E.H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York - Amsterdam, Artificial Intelligence Series, Vol. 2, 1976.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.

[Sodhi 91]
J. Sodhi, *Software Engineering Methods, Management, and CASE Tools*, TAB Professional and Reference Books, Blue Ridge Summit, PA, 1991.

[Soloway et al. 88]
E. Soloway, J. Pinto, S. Letovsky, D. Littman, R. Lampert, *Designing Documentation to Compensate for Delocalized Plans*, Communications of the ACM, Vol. 31, No. 11, November 1988, pp. 1259-1267.

[Soloway, Ehrlich 84]
E. Soloway, K. Ehrlich, *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984.

[Stahl 92]
G. Stahl, *Toward a Theory of Hermeneutic Software Design*, Technical Report CU-CS-589-92, Department of Computer Science, University of Colorado, Boulder, CO, March 1992.

[Standish 84]
T.A. Standish, *An Essay on Software Reuse*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 494-497.

[Suchman 87]
L.A. Suchman, *Plans and Situated Actions*, Cambridge University Press, Cambridge, UK, 1987.

[Sumner et al. 91]
T. Sumner, S. Davies, A.C. Lemke, P.Polson, *Iterative Design of a Voice Dialog Design Environment*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1991.

[Swartout 83]
W.R. Swartout, *XPLAIN: A System for Creating and Explaining Expert Consulting Programs*, ISI Reprint Series ISI/RS-83-4, Information Sciences Institute, University of Southern California, Marina del Rey, CA, July 1983.

[Symbolics 88]
Symbolics, Inc., *Symbolics Documentation: Concordia*, Cambridge, MA, 1988.

[Symbolics 91]
Symbolics, Inc., *Common Lisp Interface Manager (CLIM): Release 1.0*, Burlington, MA, 1991.

[Tou et al. 82]
F.N. Tou, M.D. Williams, R.E. Fikes, A. Henderson, T.W. Malone, *RABBIT: An Intelligent Database Assistant*, Proceedings of AAAI-82, Second National Conference on Artificial Intelligence (Pittsburgh, PA), Morgan Kaufmann, Los Altos, CA, August 1982, pp. 314-318.

[Tracz 88]
W. Tracz, *Software Reuse Myths*, ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 1, Jan 1988, pp. 17-21.

[Ullman 82]
J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD, 1982.

[White 91]
D.A. White, *The Knowledge-Based Software Assistant: A Program Summary*, Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY), Rome Laboratory, Griffiss AFB, New York, September 1991, pp. vi-xiii, (in press as an IEEE publication).

[Woods 85]
W.A. Woods, *What's in a Link*, in R.J. Brachman, H.J. Levesque (eds.), *Readings in Knowledge Representation*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985, pp. 218-241.

Appendix I. Excerpts of Code

This appendix provides the code for three methods that implment the search described in Chapter 4. The first two functions support the main search method that comes at the end.

```
;;; concept-match-quick-body
;;;
;;;   How to recognize concepts.  e.g., (concept-match defun-203 '(defun
;;;   cl-concept)) is true if defun-203 has a concept id of 'defun and is in
;;;   perspective 'cl-concept.  (later add possibility for alternatives).
;;;
;;;   define as macros, so they can be included in appropriate method definitions
;;;   (more efficient?).

(defmacro concept-match-quick-body (object type)
  `(with-slots (perspective id) ,object
     (let* ((want-id (if (consp ,type) (first ,type) ,type))
            (want-perspective (if (consp ,type) (or (second ,type) t) t))
            (want-source (if (consp ,type) (or (third ,type) t) t)))
       (and
        ;; match id
        (or (eq want-id t)
            (if (consp want-id)
                (if (eq (first want-id) :not)
                    (not (member id (rest want-id)))
                    (member id want-id))
                (eq id want-id)))
        ;; match perspective
        (or (eq want-perspective t)
            (if (consp want-perspective)
                (if (eq (first perspective) :not)
                    (not (member perspective (rest want-perspective)))
                    (member perspective want-perspective))
                (eq perspective want-perspective)))
        ;; match type
        (if (consp want-source)
            (if (eq (first want-source) :not)
                (not (typep ,object (cons 'or (rest want-source))))
                (typep ,object (cons 'or want-source)))
            (typep ,object want-source))))))

(defmethod concept-match-quick ((object concept) type)
  (concept-match-quick-body object type))

;;; find-candidates

;;;   possible ways to branch out from a concept are through the slots:
;;;   perspectives, parts, or roles.  The how parameter dicticts what
;;;   combinations apply:

;;;     :generalize     my perspectives and my roles
;;;     :specialize     my perspectives and my parts
;;;     :self           my perspectives
;;;     :any            my perspectives, parts, and roles

;;; NOTE:  does not return self.  this one is intended for use primarily by the
;;; search function paths-to-concept which is checking self value separately.

(defmethod find-candidates ((self concept) how)
  ;; how = :any [way], :self, :generalize, or :specialize
  (let ((slots-to-check (cdr (assoc how
                                    '((:any perspectives parts roles)
                                      (:self perspectives)
                                      (:specialize perspectives parts)
                                      (:generalize perspectives roles))))))
    (mapcan #'(lambda (slot) (and (slot-boundp self slot)
                                  (consp (slot-value self slot))        ;avoid literals, etc.
                                  (copy-list (slot-value self slot))))
            slots-to-check)))

;;; paths-to-concept
;;;
;;;   "reasoning" is in part establishing the right connections between concepts.
;;;
;;;   "breadth-first" search out paths from a starting path to a concept of a
;;;   desired type.  direction for branching out the search depends on the how
;;;   parameter
;;;
;;;   searches may be for concepts of a certain perspective or simple concepts.

;;;   NOTE:
;;;
;;;   [1] were using just a (typep lead-concept 'fragment) here but got compile
;;;   error because that type is not compiled until next file in system.lisp.

(defmethod paths-to-concepts ((self concept) to-type
                              &optional (how :any) (immediate t) (distance nil) (time nil)
                              (shifts nil) (trace nil))
  (do ((how-slots (rest
                   (assoc how '((:any perspectives parts roles)
                                (:self perspectives)
                                (:specialize perspectives parts)
```

```
                              (:generalize perspectives roles)))))
     (search-paths (list (list 'self self)))
     (found-paths nil)
     (iterations 0 (1+ iterations))
     (start-time (get-internal-run-time))
     (elapsed-time 0 (- (get-internal-run-time) start-time))
     (too-long (and time (* time internal-time-units-per-second)))
     (high-count 100) (too-many nil))
    ((or (null search-paths)
         (and immediate found-paths)
         (and distance (>= iterations distance))
         (and time (>= elapsed-time too-long)))
     (values (nreverse found-paths)
             iterations
             (float (/ elapsed-time internal-time-units-per-second))))
  ;;
  ;; check paths -- matches go to found list --  mismatches go to keep checking
  ;;
  ;; discard paths if there have been too many perspective shifts - even
  ;; before checking if they are solutions.
  ;;
  ;; also, discard paths ending in fragments:  they should be checked (as we
  ;; will) but they cannot be used for extending the search (no parts or
  ;; roles).  fragments would be represented anyway by concepts we encounter.
  ;;
  ;; also, when there are too many paths, stop expanding through perspectives
  ;;
  (do* ((paths search-paths (rest paths))
        (path (first paths) (first paths))
        (lead-concept (second path) (second path))
        (kind-concept (first path) (first path))
        (new-search-paths nil))
       ((null paths) (setf search-paths new-search-paths))
    (cond ((and shifts (> (count 'perspectives path) shifts)))
          ((concept-match-quick lead-concept to-type) (push path found-paths))
          ((concept-match-quick lead-concept '(t t fragment)))        ; [1]
          ((and too-many (eq kind-concept 'perspectives)))
;;;       ((> (count 'perspectives path) 1)) ;above ~ same but less costly
          (t (push path new-search-paths))))
  ;;
  ;; augment search paths as specified by "how" but discard those that can't
  ;; be extended (no candidates), those that cycle, those that start to
  ;; intersect.
  ;;
  ;; also, we need to record the type of candidate for potential discard (see
  ;; above).
  ;;
  (do* ((paths search-paths (rest paths))
        (path (first paths) (first paths))
        (lead-concept (second path) (second path))
        (new-search-paths nil))
       ((null paths) (setf search-paths
                           (delete-duplicates new-search-paths :key #'second)))
    (dolist (slot how-slots)
      ;; note: (parts x) can be an atom ...
      (dolist (candidate (if (consp (slot-value lead-concept slot))
                             (slot-value lead-concept slot)))
        (unless (member candidate path)
          (push (cons slot (cons candidate path)) new-search-paths)))))
  ;;
  ;; for controlling expansion (see above)
  ;;
  (unless too-many
    (setf too-many (> (length search-paths) high-count)))
  ;;
  ;; trace extensions
  (when trace
    (print (list iterations
                 (floor (/ elapsed-time internal-time-units-per-second))
                 (length search-paths) (length found-paths)))
    (terpri) (terpri))
  ))
```

Appendix II. Detailed Information for the Formal Experiment

This appendix provides additional information relevant to the formal experiment described in Chapter 6.

```
(defun cyclic-group-100 ()
  (let ((2pi (* 2.0 pi)) (pi/2 (/ pi 2.0)) (radius 30.0))
    (graphics:with-room-for-graphics (*standard-output* 200)
      (graphics:with-graphics-translation (*standard-output* 150 100)
        (graphics:with-graphics-scale (*standard-output* 2.25)
          (graphics:draw-circle 0 0 radius :filled nil)
```

Alteration Point A - Make increments of 12 not 10

```
          (do* ((theta-incr (/ 2pi 10.0))
                (x)
                (y)
```

Alteration Point B - Make a total of 12 increments

```
                (theta-list (list 0.0
                                  theta-incr
                                  (* 2.0 theta-incr)
                                  (* 3.0 theta-incr)
                                  (- 2pi (* 2.0 theta-incr))
                                  (- 2pi theta-incr))
                            (cdr theta-list))
```

Alteration Point C - Replace with labels for clock numerals

```
                (label-list '("e = 0" "1" "2" "3" "98" "99")
                            (cdr label-list))
                (theta (car theta-list) (car theta-list))
                (label (car label-list) (car label-list))
                (x-attachment))
               ((null theta-list) nil)
```

Alteration Point D - Delete, no x's needed on clock face

```
             (setq x (* radius (cos (- theta pi/2)))
                   y (* radius (sin (- theta pi/2))))
             (graphics:draw-string "x"
                                   x
                                   y
                                   :attachment-x
                                   :center
                                   :attachment-y
                                   :center)
```

Alteration Point E - Make clock numerals fall inside the circle - Replace + with -

```
             (setq x (* (+ radius 5) (cos (- theta pi/2)))
                   y (* (+ radius 5) (sin (- theta pi/2))))
```

Alteration Point F - Center clock numerals - Change attachment-x to :center

```
             (setq x-attachment (cond ((< x 0) :right)
                                      ((= (floor x) 0) :center)
                                      (t :left)))
             (graphics:draw-string label
                                   x
                                   y
                                   :attachment-x
                                   x-attachment
                                   :attachment-y
                                   :center)))))))
```

Figure II-1: Annotated Listing of the Cyclic Group Example

Subjects were expected to attend to these specific points in the example and modify them to create the clock picture.

```
(defun cyclic-group-100 ()
  (let ((2pi (* 2.0 pi)) (pi/2 (/ pi 2.0)) (radius 30.0))
    (graphics:with-room-for-graphics (*standard-output* 200)
      (graphics:with-graphics-translation (*standard-output* 150 100)
        (graphics:with-graphics-scale (*standard-output* 2.25)
          (graphics:draw-circle 0 0 radius :filled nil)
```

Alteration Point A - 3rd - 21 min

```
          (do* ((theta-incr (/ 2pi 12.0))
```

```
                (x)
                (y)
```

Alteration Point B - 2nd - 18 min, 22 min

```
                (theta-list (list 0.0
                                  theta-incr
                                  (* 2.0 theta-incr)
                                  (* 3.0 theta-incr)
                                  (* 4.0 theta-incr)
                                  (* 5.0 theta-incr)
                                  (* 6.0 theta-incr)
                                  (* 7.0 theta-incr)
                                  (* 8.0 theta-incr)
                                  (* 9.0 theta-incr)
                                  (- 2pi (* 2.0 theta-incr))
                                  (- 2pi theta-incr))
```

```
                            (cdr theta-list))
```

Alteration Point C - 1st - 6 min, 18 min, 21 min

```
                (label-list '("6" "5" "4" "3" "2" "1" "12" "11" "10" "9" "8" "7"
```

```
                            (cdr label-list))
               (theta (car theta-list) (car theta-list))
               (label (car label-list) (car label-list))
               (x-attachment))
              ((null theta-list) nil)
```

Alteration Point D - 4th - 33 min - Deleted Code Not Shown

Alteration Point E - 5th - 34 min, 44 min

```
            (setq x (* (- radius 7) (cos (- theta pi/2)))
                  y (* (- radius 7) (sin (- theta pi/2))))
```

Alteration Point F - N/A

```
            (setq x-attachment (cond ((< x 0) :right)
                                     ((= (floor x) 0) :center)
                                     (t :left)))
            (graphics:draw-string label
                                  x
                                  y
                                  :attachment-x
                                  x-attachment
                                  :attachment-y
                                  :center)))))))
```

Figure II-2: Annotated Solution of the Clock Task

Subject #23 visited the alteration points in the order indicated by the box labels. The last timing is the time at which the alteration was made correctly.

SUBJECTS USED EXPLAINER

#	Points Visited	Points Completed	Mistakes	C	R	S	T
1	(C A B)	(C A B)	NIL	4	3	28	18
2	(C D E A B)	(D A C B E)	NIL	9	4	34	16
5	(E D C A B)	(D E A B C)	NIL	10	7	49	43
10	(D A C B E)	(D A C B E)	NIL	6	4	70	64
11	(D C A B E F)	(D C A B E)	(γ FF)	9	5	46	32
18	(C A B D F E)	(C A D B E F)	NIL	11	5	36	31
19	(C B A E)	(C B A E)	(α)	5	5	47	45
23	(C B A D E)	(A C B D E)	NIL	9	7	44	38
mean, no 0's				7.88	5.00	44.25	35.88
median				9.0	5.0	45.0	35.0
mode(freq's)				9(3)	5(3)	(0)	(0)
deviation				2.53	1.41	12.73	15.51
dev./mean %				32.15	28.28	28.76	43.22

SUBJECTS USED EXPLAINER-OFF

#	Points Visited	Points Completed	Mistakes	C	R	S	T
4	(C A D E B)	(C A D B E)	NIL	11	7	40	18
7	(C A B D E)	(C A B D E)	NIL	13	11	57	8
8	(C B)	(C B)	NIL	3	3	31	10
9	(A C B E D)	(A C B E D)	NIL	6	3	30	24
13	(C B A D E)	(A B C D E)	(δ α)	23	19	46	2
14	(C A D B E F)	(C A D E B F)	NIL	11	5	61	47
15	(A B C E D)	(A E C D B)	(β)	9	3	33	29
21	(C A B D E)	(C A B D E)	NIL	6	3	39	37
mean, no 0's				10.25	6.75	42.13	21.88
median				10.0	4.0	39.5	21.0
mode(freq's)				6(2)	3(4)	(0)	(0)
deviation				6.11	5.70	11.72	15.39
dev./mean %				59.63	84.46	27.81	70.37

SUBJECTS USED DOCUMENT-EXAMINER

#	Points Visited	Points Completed	Mistakes	C	R	S	T
6	(D A B C E)	(D A B C E)	NIL	6	2	15	0
12	NIL	NIL	NIL	0	0	26	16
16	(C A B D F E)	(C A D B E F)	NIL	17	13	30	7
17	(A B C E D F)	(A B C E D F)	NIL	9	7	41	1
20	(C A B E D F)	(A D B C E)	(FF)	16	8	48	9
22	(C B A E D F)	(C A B D E F)	(β)	25	16	41	18
24	(C B A)	(C B A)	(γ α)	27	28	74	14
25	(C D A B E)	(C D A B E)	(α)	8	5	52	34
mean, no 0's				15.43	11.29	40.88	14.14
median				16.5	10.5	41.0	11.5
mode(freq's)				(0)	(0)	41(2)	(0)
deviation				8.30	8.75	18.05	10.51
dev./mean %				53.82	77.54	44.16	74.32

Table II-1: Summary of Basic Data

SUBJECTS USED EXPLAINER

# Points Completed	C	C′	R	R′	S	S′	T	T′
1 (C A B)	4	4	3	3	28	28	18	18
2 (D A C B E)	9	9	4	4	34	34	16	16
5 (D E A B C)	10	10	7	7	49	49	43	43
10 (D A C B E)	6	6	4	4	70	70	64	64
11 (D C A B E)	9	8	5	4	46	43	32	32
18 (C A D B E F)	11	8	5	4	36	31	31	28
19 (C B A E)	5	5	5	5	47	47	45	45
23 (A C B D E)	9	9	7	7	44	44	38	38
mean, no 0′s	7.88	7.38	5.00	4.75	44.25	43.25	35.88	35.50
median	9.00	8.00	5.00	4.00	45.00	43.50	35.00	35.00
mode(freq′s)	9(3)	8(2)	5(3)	4(4)	(0)	(0)	(0)	(0)
deviation	2.53	2.13	1.41	1.49	12.73	13.26	15.51	15.68
dev./mean %	32.15	28.93	28.28	31.33	28.76	30.67	43.22	44.16

SUBJECTS USED EXPLAINER-OFF

# Points Completed	C	C′	R	R′	S	S′	T	T′
4 (C A D B E)	11	11	7	7	40	40	18	18
7 (C A B D E)	13	13	11	11	57	57	8	8
8 (C B)	3	3	3	3	31	31	10	10
9 (A C B E D)	6	6	3	3	30	30	24	24
13 (A B C D E)	23	23	19	19	46	46	2	2
14 (C A D E B F)	11	7	5	3	61	36	47	24
15 (A E C D B)	9	9	3	3	33	33	29	29
21 (C A B D E)	6	6	3	3	39	39	37	37
mean, no 0′s	10.25	9.75	6.75	6.50	42.13	39.00	21.88	19.00
median	10.00	8.00	4.00	3.00	39.50	37.50	21.00	21.00
mode(freq′s)	6(2)	6(2)	3(4)	3(5)	(0)	(0)	(0)	24(2)
deviation	6.11	6.20	5.70	5.83	11.72	8.98	15.39	11.75
dev./mean %	59.63	63.64	84.46	89.71	27.81	23.02	70.37	61.83

SUBJECTS USED DOCUMENT-EXAMINER

# Points Completed	C	C′	R	R′	S	S′	T	T′
6 (D A B C E)	6	6	2	2	15	15	0	0
12 NIL	0	0	0	0	26	26	16	16
16 (C A D B E F)	17	9	13	8	30	17	7	1
17 (A B C E D F)	9	6	7	4	41	23	1	0
20 (A D B C E)	16	15	8	7	48	36	9	9
22 (C A B D E F)	25	21	16	13	41	36	18	18
24 (C B A)	27	27	28	28	74	74	14	14
25 (C D A B E)	8	8	5	5	52	52	34	34
mean, no 0′s	15.43	13.14	11.29	9.57	40.88	34.88	14.14	15.33
median	16.50	12.00	10.50	7.50	41.00	31.00	15.00	17.00
mode(freq′s)	(0)	6(2)	(0)	(0)	41(2)	36(2)	(0)	0(2)
deviation	8.30	8.19	8.75	8.85	18.05	19.86	10.51	10.98
dev./mean %	53.82	62.35	77.54	92.44	44.16	56.95	74.32	71.64

Table II-2: Summary and Comparison of Revised Data

Data affecting variables as a result of a subject's work on Alteration Point F was removed to make for a fairer comparison (see Section 6.6.2). The revised values are presented with the previous values for the reader.

SUBJECTS USED EXPLAINER

# Points Completed	C/G	C'/G'	R/G	R'/G'	S/G	S'/G'	T/S %	T'/S' %
1 (C A B)	1.33	1.33	1.00	1.00	9.33	9.33	64.29	64.29
2 (D A C B E)	1.80	1.80	0.80	0.80	6.80	6.80	47.06	47.06
5 (D E A B C)	2.00	2.00	1.40	1.40	9.80	9.80	87.76	87.76
10 (D A C B E)	1.20	1.20	0.80	0.80	14.00	14.00	91.43	91.43
11 (D C A B E)	1.80	1.60	1.00	0.80	9.20	8.60	69.57	69.57
18 (C A D B E F)	1.83	1.60	0.83	0.80	6.00	6.20	86.11	86.11
19 (C B A E)	1.25	1.25	1.25	1.25	11.75	11.75	95.74	95.74
23 (A C B D E)	1.80	1.80	1.40	1.40	8.80	8.80	86.36	86.36
mean, no nil's	1.63	1.57	1.06	1.03	9.46	9.41	78.54	78.54
median, nil->+inf	1.80	1.60	1.00	0.90	9.27	9.07	86.24	86.24
mode(freq's)	1.80(3)	1.60(2)	1.40(2)	0.80(4)	(0)	(0)	(0)	(0)
deviation	0.31	0.29	0.26	0.28	2.55	2.53	16.65	16.65
dev./mean %	19.19	18.42	24.20	26.80	26.96	26.90	21.20	21.20

SUBJECTS USED EXPLAINER-OFF

# Points Completed	C/G	C'/G'	R/G	R'/G'	S/G	S'/G'	T/S %	T'/S' %
4 (C A D B E)	2.20	2.20	1.40	1.40	8.00	8.00	45.00	45.00
7 (C A B D E)	2.60	2.60	2.20	2.20	11.40	11.40	14.04	14.04
8 (C B)	1.50	1.50	1.50	1.50	15.50	15.50	32.26	32.26
9 (A C B E D)	1.20	1.20	0.60	0.60	6.00	6.00	80.00	80.00
13 (A B C D E)	4.60	4.60	3.80	3.80	9.20	9.20	4.35	4.35
14 (C A D E B F)	1.83	1.40	0.83	0.60	10.17	7.20	77.05	77.05
15 (A E C D B)	1.80	1.80	0.60	0.60	6.60	6.60	87.88	87.88
21 (C A B D E)	1.20	1.20	0.60	0.60	7.80	7.80	94.87	94.87
mean, no nil's	2.12	2.06	1.44	1.41	9.33	8.96	54.43	54.43
median, nil->+inf	1.82	1.65	1.12	1.00	8.60	7.90	61.02	61.02
mode(freq's)	1.20(2)	1.20(2)	0.60(3)	0.60(4)	(0)	(0)	(0)	(0)
deviation	1.11	1.14	1.11	1.13	3.06	3.13	35.13	35.13
dev./mean %	52.51	55.16	77.05	80.14	32.83	34.90	64.55	64.55

SUBJECTS USED DOCUMENT-EXAMINER

# Points Completed	C/G	C'/G'	R/G	R'/G'	S/G	S'/G'	T/S %	T'/S' %
6 (D A B C E)	1.20	1.20	0.40	0.40	3.00	3.00	0.00	0.00
12 NIL	-.--	-.--	-.--	-.--	-.--	-.--	61.54	61.54
16 (C A D B E F)	2.83	1.80	2.17	1.60	5.00	3.40	23.33	23.33
17 (A B C E D F)	1.50	1.20	1.17	0.80	6.83	4.60	2.44	2.44
20 (A D B C E)	3.20	3.00	1.60	1.40	9.60	7.20	18.75	18.75
22 (C A B D E F)	4.17	4.20	2.67	2.60	6.83	7.20	43.90	43.90
24 (C B A)	9.00	9.00	9.33	9.33	24.67	24.67	18.92	18.92
25 (C D A B E)	1.60	1.60	1.00	1.00	10.40	10.40	65.38	65.38
mean, no nil's	3.36	3.14	2.62	2.45	9.48	8.64	33.47	33.47
median, nil->+inf	3.02	2.40	1.88	1.50	8.22	7.20	21.13	21.13
mode(freq's)	(0)	1.20(2)	(0)	(0)	6.83(2)	7.20(2)	(0)	(0)
deviation	2.71	2.80	3.05	3.12	7.16	7.52	23.84	23.84
dev./mean %	80.65	89.22	116.63	127.31	75.55	87.11	71.23	71.23

Table II-3: Summary of Ratio Data

SUBJECTS USED EXPLAINER

#	Points Completed	ChangesA	ChangesB	ChangesC	ChangesD	ChangesE	ChangesF	Total
1	(C A B)	1	2	1	0	0	0	4
2	(D A C B E)	1	2	2	1	3	0	9
5	(D E A B C)	4	1	2	1	2	0	10
10	(D A C B E)	1	1	1	1	2	0	6
11	(D C A B E)	1	1	1	1	4	1	9
18	(C A D B E F)	1	2	1	1	3	3	11
19	(C B A E)	1	1	1	0	2	0	5
23	(A C B D E)	1	2	3	1	2	0	9
mean, no 0's		1.38	1.50	1.50	1.00	2.57	2.00	7.88
median, 0->+inf		1.00	1.50	1.00	1.00	2.50	+INF	9.00
mode(freq's)		1(7)	1(4)	1(5)	1(6)	2(4)	0(6)	9(3)
deviation		1.06	0.53	0.76	0.00	0.79	1.41	2.53
dev./mean %		77.14	35.63	50.40	0.00	30.60	70.71	32.15

SUBJECTS USED EXPLAINER-OFF

#	Points Completed	ChangesA	ChangesB	ChangesC	ChangesD	ChangesE	ChangesF	Total
4	(C A D B E)	1	5	1	1	3	0	11
7	(C A B D E)	1	5	1	1	5	0	13
8	(C B)	0	2	1	0	0	0	3
9	(A C B E D)	1	1	2	1	1	0	6
13	(A B C D E)	1	6	6	1	9	0	23
14	(C A D E B F)	1	3	1	1	1	4	11
15	(A E C D B)	1	3	2	1	2	0	9
21	(C A B D E)	1	1	1	1	2	0	6
mean, no 0's		1.00	3.25	1.88	1.00	3.29	4.00	10.25
median, 0->+inf		1.00	3.00	1.00	1.00	2.50	+INF	10.00
mode(freq's)		1(7)	3(2)	1(5)	1(7)	2(2)	0(7)	6(2)
deviation		0.00	1.91	1.73	0.00	2.87	0.00	6.11
dev./mean %		0.00	58.73	92.10	0.00	87.35	0.00	59.63

SUBJECTS USED DOCUMENT-EXAMINER

#	Points Completed	ChangesA	ChangesB	ChangesC	ChangesD	ChangesE	ChangesF	Total
6	(D A B C E)	1	1	1	1	2	0	6
12	NIL	0	0	0	0	0	0	0
16	(C A D B E F)	1	5	1	1	1	8	17
17	(A B C E D F)	1	1	1	1	2	3	9
20	(A D B C E)	1	4	5	1	4	1	16
22	(C A B D E F)	2	14	1	1	3	4	25
24	(C B A)	9	13	5	0	0	0	27
25	(C D A B E)	1	2	1	1	3	0	8
mean, no 0's		2.29	5.71	2.14	1.00	2.50	4.00	15.43
median, 0->+inf		1.00	4.50	1.00	1.00	3.00	8.00	16.50
mode(freq's)		1(5)	1(2)	1(5)	1(6)	3(2)	0(4)	(0)
deviation		2.98	5.53	1.95	0.00	1.05	2.94	8.30
dev./mean %		130.55	96.76	91.08	0.00	41.95	73.60	53.82

Table II-4: Summary of Changes by Individual Alteration Points

Post Experiment Questionnaire for the Experiment Example-based Software Reuse — DocEx

This questionnaire is to be filled out by the experiment leader and a subject **after** *the subject has participated in the experiment titled Example-based Software Reuse. This form is accompanied by a Statement of Informed Consent.*

Rating of Document Examiner Tool

Overall?	hated it	felt it okay	liked it	liked it a lot
Interface?	hated it	felt it okay	liked it	liked it a lot
Information?	not useful	felt it okay	useful	very useful

Rating of Documentation

Syntactic Summary	disliked it	felt it okay	liked it	liked it a lot
Text Description	disliked it	felt it okay	liked it	liked it a lot
Code Example	disliked it	felt it okay	liked it	liked it a lot
Preferred Information (1 = best, 4 = least)	__ Syntax __ Document Examples		__ Text __ Task Example	

Rating of Task and Solution

Liked Task?	disliked it	felt it okay	liked it	liked it a lot
Liked Example?	hated it	felt it okay	liked it	liked it a lot
Satisfied with solution?	not satisfied	somewhat satisfied	satisfied	very satisfied

Questions (Answer 1-2 sentences)

What was the most useful information the document examiner gave you (in general or specific instance)?

What was the least useful aspect of the document examiner?

What did you feel the document examiner lacked that you felt you wanted to do or ask?

Any comments on ratings given above.

Figure II-3: Post Experiment Questionnaire followed by Interviewer

This form served as a guide for the Experimenter. Like and dislike were explained to the subjects as including usefulness/helpfullness and useless/not helpful, with respect to the task the subjects were trying to solve. This questionnaire was used for the DOCUMENTEXAMINER Group.

Post Experiment Questionnaire for the Experiment Example-based Software Reuse - Explainer

This questionnaire is to be filled out by the experiment leader and a subject **after** *the subject has participated in the experiment titled Example-based Software Reuse. This form is accompanied by a Statement of Informed Consent.*

Rating of Explainer Tool

Overall?	hated it	felt it okay	liked it	liked it a lot
Interface?	hated it	felt it okay	liked it	liked it a lot
Information?	not useful	felt it okay	useful	very useful

Rating of Explanation

Code Pane	hated it	felt it okay	liked it	liked it a lot
Example Pane	hated it	felt it okay	liked it	liked it a lot
Diagram Pane	hated it	felt it okay	liked it	liked it a lot
Text Pane	hated it	felt it okay	liked it	liked it a lot
Preferred Information (1 = best, 4 = least)	__ Code	__ Example	__ Diagram	__ Text

Rating of Task and Solution

Liked Task?	hated it	felt it okay	liked it	liked it a lot
Liked Example?	hated it	felt it okay	liked it	liked it a lot
Satisfied with solution?	not satisfied	somewhat satisfied	satisfied	very satisfied

Questions (Answer 1-2 sentences)

What was the most useful information the Explainer Tool gave you (in general or specific instance)?

What was the least useful aspect of the Explainer tool?

What did you feel the Explainer tool lacked that you felt you wanted to do or ask?

Any comments on ratings given above.

Figure II-4: Post Experiment Questionnaire followed by Interviewer

This questionnaire was used by the Experimenter for both Full and Menu-Off Explainer Groups.