Supporting Software Designers with Integrated Domain-Oriented Design Environments

Gerhard Fischer, Andreas Girgensohn, Kumiyo Nakakoji, and David Redmiles

Abstract- **The field of knowledge-based software engineering has been undergoing a shift in emphasis from automatic programming to human augmentation. We support this shift with an approach that embeds human-computer cooperative problem-solving tools into knowledge-based design environments that work in conjunction with human software designers in specific application domains. Domain orientation reduces the large conceptual distance between problem-domain semantics and software artifacts. Integrated environments support the coevolution of specification and construction while allowing designers to access relevant knowledge at each stage of a software development process. The access and development of knowledge is supported in a cycle of location, comprehension, and modification. Modification includes the evolution of the knowledge base and tools. A framework for building such tools and mechanisms is described and illustrated in terms of three systems: CATALOGEXPLORER, EXPLAINER, and MODIFIER. User studies of these systems demonstrate the promises and limitations of our design environment approach.**

Index Terms— **Coevolution of specification and construction, domain orientation, end-user modifiability, explanation, information retrieval, knowledge representation, knowledge-based design environments, software reuse and redesign, user interface.**

I. INTRODUCTION

THE field of knowledge-based software engineering has been undergoing a shift in emphasis from automatic programming to human augmentation. A growing number of researchers are using knowledge-based systems and new communication paradigms to assist software engineers, not to replace them. The idea of human augmentation, beginning with Engelbart [12], has gained strength steadily through now-familiar projects such as the Knowledge-Based Software Assistant [55], the Programmer's Apprentice [54], the Software Designer's Associate [27], the Knowledge Base Designer's Assistant [43], and earlier systems described by Barstow, Shrobe, and Sandewall [3].

Parallel to these projects, the need for domain orientation has received increased recognition. The software reuse community is concerned with approaches to domain modeling [38]. The "thin spread of application knowledge" has been identified in empirical studies [10]. Just as good writers cannot write books on subjects with which they are not familiar, good software engineers cannot write programs for domains they do not understand. The goal of knowledge-based software tools is to shorten the large conceptual distance between problem domain semantics and software artifacts [18].

Many knowledge-based tools have been built around design artifacts with little emphasis on how to support human aspects. Our approach goes beyond these systems by embedding cooperative problem-solving tools into knowledge-based design environments that work in conjunction with human software designers in specific application domains [19]. These design environments imply a separation of software engineers into two types: those who build domain-oriented software environments using generic programming languages, and those who build application software programs using such domain-specific software environments.

This paper will provide software engineers, acting as design environment builders, with guidance in building such domain-specific design environments. The architectures and techniques presented describe domain-independent approaches for building domain-specific design environments and suggest how such design environments can be used by software engineers acting as end users of such design environments. Critical to our approach is how this latter class of end users bears some of the burden of maintaining and increasing the domain-dependent knowledge base using end-user modification techniques. The techniques described allow end users knowledgeable in the application domain to extend the knowledge base without the assistance of the original design environment builders.

II. CONCEPTUAL FRAMEWORK

Software design, use, and maintenance is best understood as an evolutionary design process [45]. As such, it shares many features with other, more established design disciplines, such as architectural design, engineering design, musical composition, and writing. Based on an analysis of the historical development of design methods in these other disciplines, Archer [1, p. 348] observed that "one of the features of the early theories of design methods that really disenchanted many practicing designers was their directionality and causality and separation of analysis from synthesis, all of which was perceived by the designers as being unnatural."

A field study by our research group supported the above observation [21]. Our study also showed that in many cases people were initially unable to articulate complete requirements and tended to start from a partial specification and refine it incrementally based on feedback. This observation

Manuscript received February 5, 1992. This work was supported by the National Science Foundation under Grants IRI-8722792 and IRI-9015441, by the Army Research Institute under Grant MDA903-86-C0143, and by Software Research Associates. Recommended by A. Borgida and M. Jarke.

The authors are with the Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder, Colorado 80309. K. Nakakoji is also affiliated with the Software Engineering Laboratory, Software Research Associates, Chiyoda-ku, Tokyo 102, Japan.

IEEE Log Number 9200169.

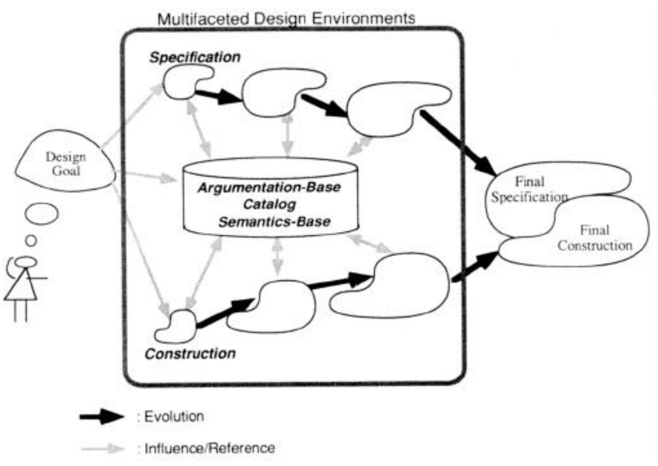


Fig. 1. Coevolution of specification and construction.

concurs with that of Bauer [4], who describes programming as an evolutionary process. Whereas Bauer interprets evolution as a process that transforms formal specifications into program code, we believe that evolution applies more generally to the development of a better understanding of a real world problem and a better mapping of that problem to a formal construct. Automated design methodologies fail to support designers in problem framing [40] and require complete representations of problem domains before starting design [2].

What is required, then, of knowledge-based and domain-oriented design tools is an integrated environment that can support the coevolution of specification and construction [49]. Furthermore, information delivered by the computer needs to be conveyed to people in ways they can comprehend, and likewise, people need convenient ways to convey their reasoning to the computer. Seeking an optimal balance between the respective capabilities and limitations of people and computers leads to the notion of cooperative problem-solving tools. The computer is capable of storing and searching large amounts of information such as instances of previous designs; the human is capable of reasoning with and inventing new designs from appropriate information. The focal point is the interface and its accommodation of the cognitive steps in the design process.

A. Coevolution of Specification and Construction

Fig. 1 illustrates the process of coevolution of specification and construction. This coevolution does not separate the *upstream activities* and *downstream activities* [49]. Hypertext systems for recording design decision processes and design rationale, such as gIBIS [8], SIBYL [30], and DESIGN RATIONALE [32], support designers in organizing the upstream activities and allow them to deal with framing problem requirements specification. However, these artifacts are isolated from constructed solutions. In contrast, in our approach, by integrating the upstream and downstream activities, designers are simultaneously articulating “what” they need to design and “how” to design it [41]. Designers create a partial specification or a partial construction from a vague design goal. As the design environment provides relevant feedback, designers develop the partial construction and specification gradually.

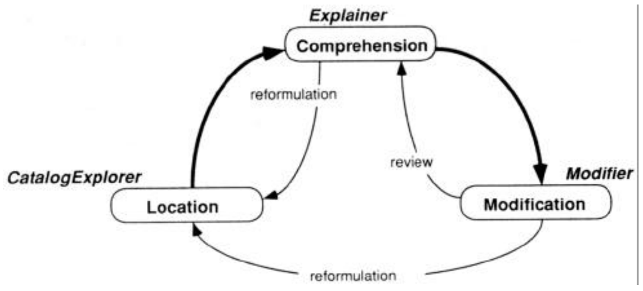


Fig. 2. Knowledge-access paradigm in design environments.

The feedback information is derived from the knowledge base consisting of argumentation, catalog of previous solutions, and domain semantics.

Both the model of coevolution of specification and construction and the spiral model of software design [7] support iteration of requirements specification and solution construction. The difference is that the spiral model places emphasis on discrete cycles of revision, remedy, and modification of design artifacts in terms of risk reduction, with a strict separation of cycles by reviews that are used to ensure the commitment to proceed to the next phase. In contrast, our model does not require any separation of the activities, but rather supports continuous growth of design, both in the specification and construction.

B. Knowledge-Access Paradigm: Location-Comprehension-Modification

Knowledge-intensive systems invariably lead to large information spaces. They contain many classes of objects, making it likely that an object close to what is needed exists. However, without adequate system support in the design process, it is difficult to locate and understand appropriate objects [16], [17], [37]. The richer the knowledge base is, the more expensive to access it, in terms of both computational and cognitive costs. Our interest is to reduce this cognitive cost in dealing with a huge amount of knowledge stored in the environment and to focus the selection and. use of knowledge around the human designer’s approach to solving a problem. Designers have to locate design information, comprehend the retrieved information, and modify it according to their current needs. Comprehension may lead directly to further retrieval, and modification may require further comprehension or additional retrieval. Fig. 2 illustrates the cycle of these activities.

Location: In a design environment, designers should be able to access knowledge relevant to the task at hand. The task at hand can be partially identified through specification. The partially identified task can be used to provide background for information retrieval processes. Using the partial specification relieves designers of the need to specify database queries explicitly, and thereby prevents them from being distracted by information retrieval activities, which are not their primary concern in designing software.

Comprehension: The comprehension phase is a bridge between the location and subsequent use (modification) of information for solving a problem. Comprehension is necessary for designers to judge the applicability of retrieved information and may lead to reformulation of their partial specifications.

One kind of information retrieved would be examples of previous design solutions. After finding a potentially useful example, the designer is expected to explore it more carefully to build an analogy between the example task and the current task. For example, in the software domain, glancing at program code may not be sufficient to determine that it produces a correlation plot. The hypothesis is that by learning about the example program, the designer learns by analogy how to program the current task. The goal of the comprehension phase is to aid the designer in building this analogy and specifically to recognize what elements of the example are applicable to solving the current task.

Modification: After a design object has been located and understood, it might be necessary to adapt it to a new use. The representation of objects allows designers to perform at least some of the adaptations on a level above that of the programming language. For example, in a design environment that supports programming for plotting graphs, concepts such as drawing coordinate axes would be represented in the design objects and could be used for manipulating the associated program code. Evolution of the design environment is supported by allowing new concepts to be added to accommodate new design objects.

III. System-Building Efforts

During the last five years, we have developed and evaluated several prototype systems of domain-oriented design environments [19],[31]. These system-building efforts helped us to define an integrated architecture for design environments consisting of tools for evolving design artifacts and knowledge bases.

This section describes three systems that act together to support the location, comprehension, and modification phases in knowledge-based design (Fig. 2). The systems operate on design objects stored in the catalog base in the domain of LISP programs for plotting data. These design objects can be reused for case-based reasoning, such as providing a solution to a new problem, evaluating and justifying decisions behind the partial specification or construction, and informing designers of possible failures [28],[46].

The CatalogExplorer system helps designers to locate prestored design solutions relevant to their task at hand as articulated with a partial specification [20]. The Explainer system helps designers in comprehending retrieved design solutions by providing various perspectives [17]. The Modifier system helps designers to modify design objects, and the design environment itself, at a level closely related to the task [22].

A. Scenario in Accessing Design Objects Stored in the Catalog Base

A simple scenario shows how the three systems would be used. A designer wants to create a program to plot the results of two persons playing one-on-one basketball. The task is first described as supported by commonly available tools; this is then contrasted with the use of the three systems.

Assuming that there is a collection of examples available, the designer has to find one that is close to his or her needs. For this purpose, tools such as the hypertext-based Symbolics document examiner [53], a database access program, or the UNIX *grep* command could be used. These require the designer to find keywords that might appear in an appropriate example. After one or more examples are located, the designer has to read through the program code and try traces or sample executions to determine whether the example will be useful, and also has to understand what parts might be extracted or modified. Finally, the appropriate program example has to be changed with a text editor.

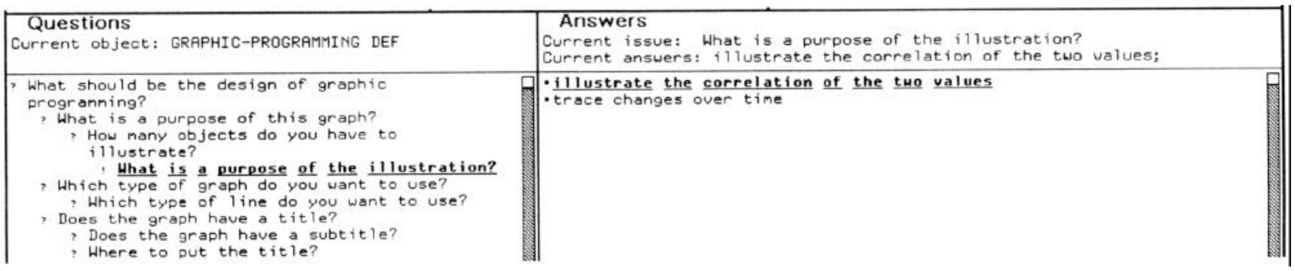
With our systems, the designer first specifies some of the requirements using the specification component provided by CatalogExplorer. CatalogExplorer then searches the stored example programs in the catalog according to the partial specification. The search yields an example program that plots the results of two people playing squash. The example code is brought into Explainer, which helps the designer understand its operation, build an analogy between the example task and the current task, and learn how to program the current task. During the examination of the retrieved example, the designer wants to add another form of representation *(perspective)* to the retrieved program, and thus invokes Modifier, which provides support to add a new perspective to the program.

B. CatalogExplorer

The CatalogExplorer system [20] provides a tool to support designers in specifying their problem requirements in terms of the problem domain. This specification partially identifies the designer's task at hand. 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 [8], [14],[29]. It consists of issues, optional answers (i.e., alternatives), and arguments to the answers. This information is accumulated by recording design rationale for design decisions made in previous design sessions. The specification component (see Fig. 3(a)) 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 a pre-defined *domain distinction.* For example, in the scenario, 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 components (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. Answers are tied to domain distinctions. Consider the example



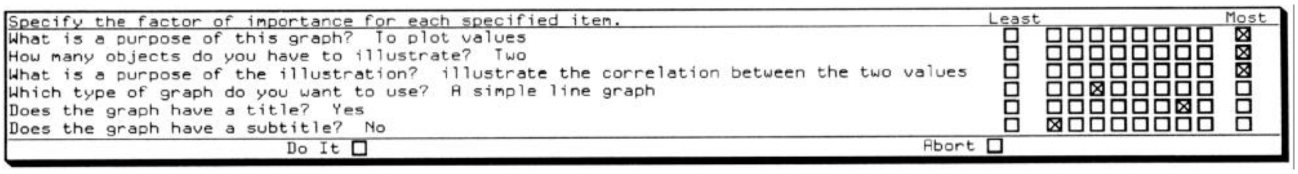
```
Questions
Current object: GRAPHIC-PROGRAMMING DEF

? 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?
```

```
Answers
Current issue:  What is a purpose of the illustration?
Current answers: illustrate the correlation of the two values;

•illustrate the correlation of the two values
•trace changes over time
```

(a)



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)

Fig. 3. Specification components. (a) Designers can specify their design requirement in a form of a questionnaire. The left window provides designers with questions. By clicking one of them, the right window provides possible answers to select. (b) After the specification, designers have to weight the importance of each specified item.

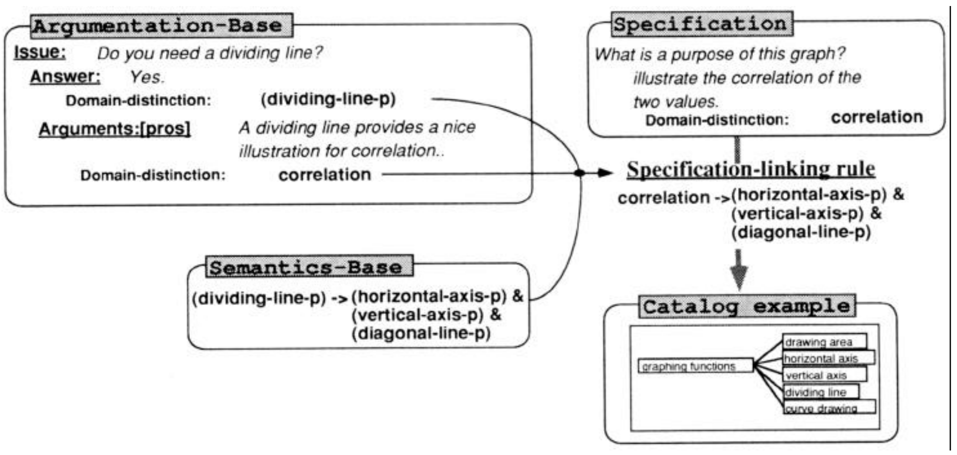

Argumentation-Base
Issue: Do you need a dividing line?
Answer: Yes.
Domain-distinction: (dividing-line-p)
Arguments:[pros] A dividing line provides a nice illustration for correlation..
Domain-distinction: correlation
Specification
What is a purpose of this graph? illustrate the correlation of the two values.
Domain-distinction: correlation
Specification-linking rule
correlation ->(horizontal-axis-p) & (vertical-axis-p) & (diagonal-line-p)
Semantics-Base
(dividing-line-p) -> (horizontal-axis-p) & (vertical-axis-p) & (diagonal-line-p)
Catalog example
graphing functions
drawing area
horizontal axis
vertical axis
dividing line
curve drawing

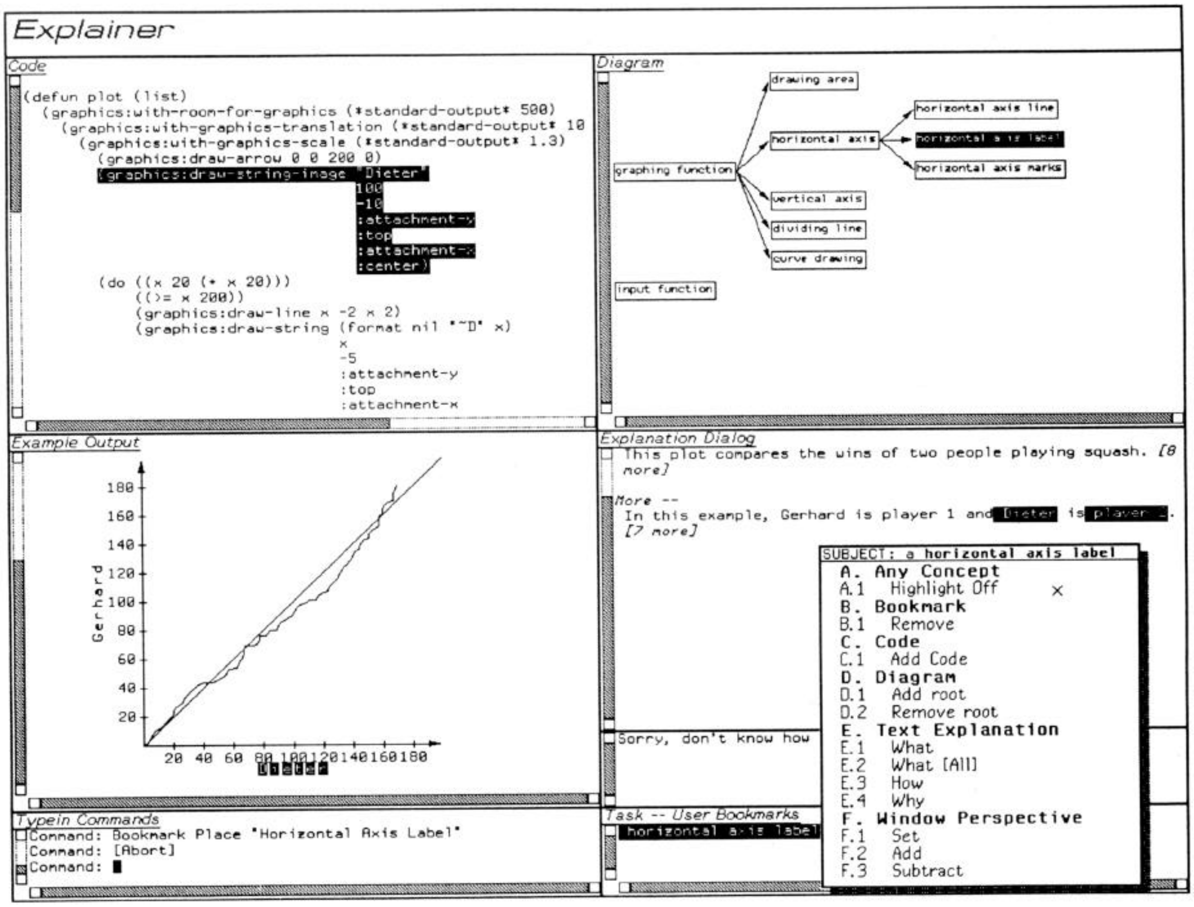
Fig. 4. Dynamic derivation of specification-linking rules in CATALOG EXPLORER. When designers specify that they want to illustrate the correlation of the two values, the system derives specification-linking rules, one of which says the illustrating correlation requires a graphic program to have a dividing line. Then, knowing that having a dividing line requires horizontal and vertical axes and a diagonal line, the design environment delivers program examples from the catalog base, each of which has horizontal and vertical-axes and a diagonal line.

illustrated in Fig. 4. The answer "Yes" to the issue "Do you need a dividing line?" is connected to the domain distinction "dividing-line-p." The pro-argument "a dividing line provides a nice illustration for correlation" has the domain distinction "correlation." Because it is a pro-argument, the dependency "correlation $\rightarrow$ dividing-line-p" is implied. Contra-arguments cause dependencies of the form "X $\rightarrow$ not Y." The specification-linking rules represent those dependencies. When designers 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 for finding 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 designers articulate specification choices, they are asked to assign a weight to each to indicate its degree of importance (see Fig. 3(b)). 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 (for details, see Fischer and Nakakoji [20]). By seeing the effects of changing the degree of importance in the ordered catalog examples, designers can make trade-offs among specification items.

C. EXPLAINER

The result of the location phase is that designers are presented with a program example for a task similar to their current task at hand. In this case (Fig. 5) designers are presented with a program that plots the results of two people



```
(defun plot (list)
  (graphics:with-room-for-graphics (*standard-output* 500)
    (graphics:with-graphics-translation (*standard-output* 10
      (graphics:with-graphics-scale (*standard-output* 1.3)
        (graphics:draw-arrow 0 0 200 0)
        (graphics:draw-string-image "Dieter"
                                    100
                                    -10
                                    :attachment-y
                                    :top
                                    :attachment-x
                                    :center)
        (do ((x 20 (+ x 20)))
            ((>= x 200))
            (graphics:draw-line x -2 x 2)
            (graphics:draw-string (format nil "~D" x)
                                  x
                                  -5
                                  :attachment-y
                                  :top
                                  :attachment-x
```

```
This plot compares the wins of two people playing squash. [8
more]

More --
In this example, Gerhard is player 1 and Dieter is player 2.
[7 more]

Sorry, don't know how
```

```
Command: Bookmark Place "Horizontal Axis Label"
Command: [Abort]
Command: █
```

Fig. 5. EXPLAINER screen. EXPLAINER provides the designer with the ability to view and explore the multiple perspectives of an example. The programming language perspective is shown in the code pane; a plotting perspective is shown in the example output pane; and a game-playing perspective is described in the explanation dialog pane.

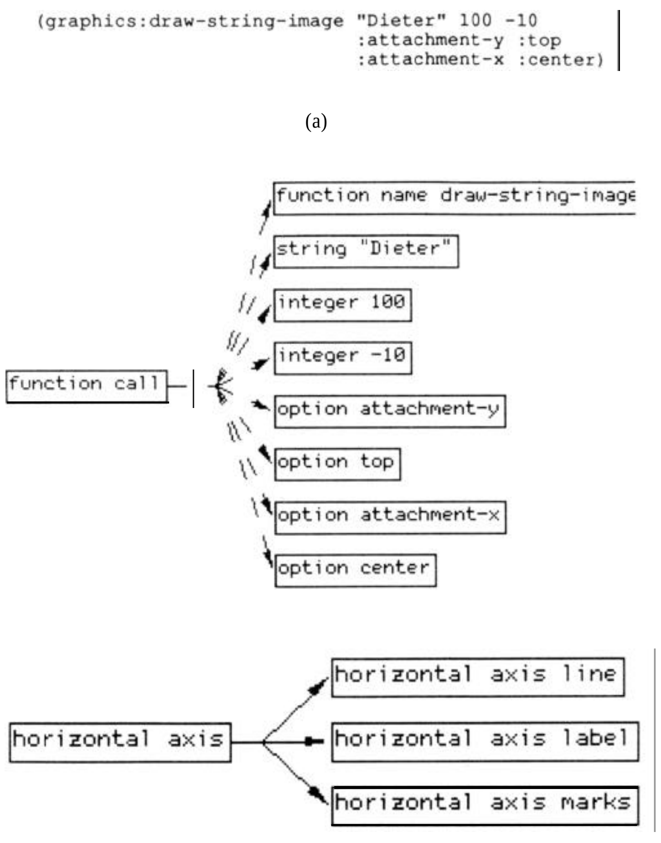
playing squash. One goal for designers in the comprehension phase is to determine an analogy between this located example and their current task of plotting basketball results. Designers are aided in building an analogy by assuming that the presented example is relevant to their current task. This foreknowledge of relevance is an implicit result of the catalog search done by CATALOGEXPLORER. The necessary details are found by exploring the example with the EXPLAINER system.

Examples are represented from multiple *perspectives;* different forms of representation, including code, diagram, sample picture, and text; and also different domains within these representations. For instance, an example may be used to illustrate programming language concepts such as function calling and looping. The same example from another perspective might illustrate graphic features such as curves or labels drawn on a plot. From still another perspective, the example could illustrate how to compare scores of two players competing in squash matches. This notion of perspectives is consistent with its use in other knowledge representation systems such as KRL [6] and MERLIN [35]. By clicking on fragments in the different perspectives and selecting actions from menus, designers can expand the information presented in the different perspectives. In Fig. 5, the designer has identified the horizontal axis label, how that plot feature is implemented in code, and how in the game-playing perspective the label is the second player.

In exploring an example from different perspectives, a minimalist strategy is followed. Designers using EXPLAINER expand only the information they determine to be relevant to their task. The use of minimal explanation in the context of an example avoids overwhelming designers with irrelevant information [13]. Actively guiding the interactions based, for example, on the requirements articulated in CATALOGEXPLORER is not yet implemented.

An important kind of knowledge supported is the mapping between different perspectives on an example. The principal point served by representing these mappings is to illustrate how elements of a task map onto features of a solution (e.g., how concepts in the squash problem map onto plot or code concepts). Thus, the usefulness of the perspective knowledge is in search algorithms to find information of the appropriate perspective and in presentation algorithms to support viewing and interaction with the knowledge.

The smallest unit of knowledge in EXPLAINER is a concept. Concepts become nodes in semantic networks. Perspectives are instantiated by collections of networks with a common theme. Fig. 6 shows two networks about the same piece of code,



```
(graphics:draw-string-image "Dieter" 100 -10
                            :attachment-y :top
                            :attachment-x :center)
```

(a)

(c)

Fig. 6. Multiple perspectives within EXPLAINER. (a) LISP code from the squash example. (b) Network of LISP concepts. (c) Network of plotting concepts.

which from the plotting perspective, draws the horizontal axis label.

The primary relations that connect concepts within a network are “is-part-of,” “consists-of,” and “is-also-a.” These relations more generally constitute generalization, specialization, and equivalence, respectively. For example, the network in Fig. 6(c) means that “a horizontal axis consists of an axis line, an axis label, and axis marks.” It also provides the information that “the axis marks are part of the horizontal axis.” An “is-also-a” link between “horizontal axis label” and “function call” (Fig. 6(b)) states that “the horizontal axis label corresponds to a function call,” though this information is more effectively presented by highlighting equivalenced objects, as shown in Fig. 5. Within a perspective, a concept can participate in several networks. Concepts within a network are always of the same perspective; only through the “is-also-a” slot are perspectives mixed.

The English text generated by EXPLAINER from these networks may vary according to the context. For example, the phrase “consists of’ is reasonable for building a sentence for the network in Fig. 6(c), but the phrase “has arguments” is more meaningful in interpreting Fig. 6(b). These variations are supported by sentence patterns associated with concepts in objects called *descriptors.* Case, number, and agreement of articles and nouns are computed according to primitive rules. There are default descriptors for concepts within a given perspective. Descriptors record presentation instructions for other views as well, e.g., how to format code. Thus, descriptors serve as a generalized notion of a lexicon in our representation scheme. Free text can also be associated with concepts in networks using the equivalence link.

A search and presentation is initiated by the designer clicking on a screen object and selecting an action from the command menu (Fig. 5). For example, suppose a designer clicks on the diagram node “horizontal axis” and selects “E.3 How” from the command menu. EXPLAINER searches for specializations of this concept and finds the three parts “axis line,” “axis label,” and “axis marks.” The default descriptor for this plotting perspective yields “consists of’ for the English pattern for specialization. Such patterns implicitly give one-to-many relations from the concept itself to nodes linked by specialization; the sentence generated by EXPLAINER is “the horizontal axis consists of an axis line, an axis label, and axis marks.”

D. MODIFIER

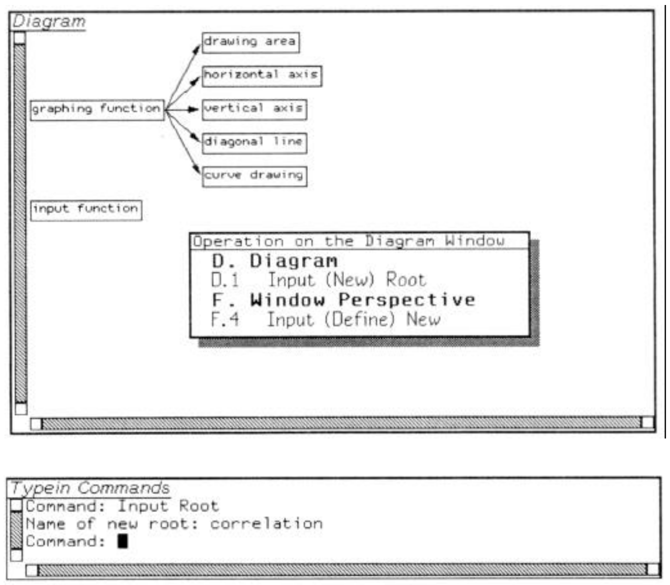
The modification component, MODIFIER, helps designers modify various kinds of objects. First, program modules found by CATALOGEXPLORER and explained by EXPLAINER need to be adapted to their use in the program being written. Second, the domain knowledge in CATALOGEXPLORER and EXPLAINER need to be modified and extended.

The modification of a program example is supported through the perspectives maintained by EXPLAINER. For example, in order to use the squash example for plotting basketball results, the spacing and values of the axes have to be changed. The corresponding LISP code can be accessed and modified through the graph of plotting concepts. A starting point is the concept “horizontal axis mark” in the “Diagram” window in Fig. 5. From there, the designer can locate loop variables and axis marks and change them as needed.

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 graph editor (see Fig. 7(a)). Concepts from different perspectives can be equated. For instance, the concept “data set 1” can be equated to the phrase “Gerhard” (see Fig. 7(b)). Concepts in many perspectives may be linked to corresponding concepts in other perspectives. The object net of concepts associated with a program example is saved together with the example. Each example is represented in terms of at least two perspectives: The LISP programming language perspective and a plotting perspective.

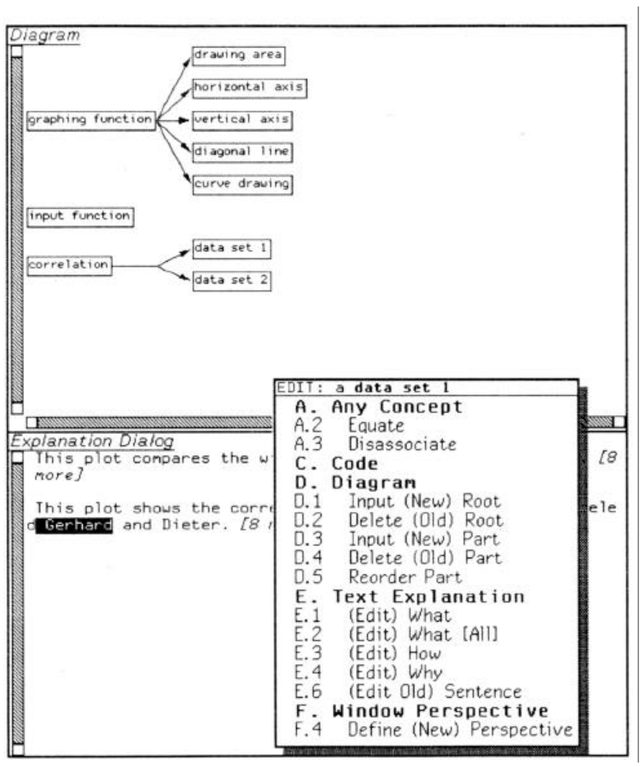
Descriptors associated with some concepts are edited in property sheets (see Fig. 8). The fields of these sheets have help displays associated with them. The help window displays all possible values, such as all program code descriptors in the system. Descriptions for any object in the help window can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

MODIFIER supports both the evolution of individual examples and the evolution of the design environment. The design environment itself evolves through the introduction of new domain knowledge, for example, adding the new



```
Command: Input Root
Name of new root: correlation
Command: ▮
```

(a)



```
This plot compares the w                                   [8
more]

This plot shows the corr                                   ele
d Gerhard and Dieter. [8 
```

(b)

Fig. 7. Adding a new concept. (a) The new concept “correlation” and its two parts “data set 1” and “data set 2” are defined with the menu command “Input Root.” (h) The concept “data set 1” is then linked to the phrase “Gerhard” in the explanation window with the menu command “Equate.”

concept “correlation” 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.

E. Mechanisms for Knowledge Representation and Reasoning

Our work is focused on creating representations supporting domain experts as end users. Therefore, our research interests in knowledge representation and reasoning are driven by the needs of these knowledge workers rather than specific properties of different formalisms. This focus and concern for efficiency led us to tailor our own representations and methods in CLOS [47] on the SYMBOLICS LISP machine. This implementation work was possible since the CATALOGEXPLORER, EXPLAINER, and MODIFIER tools did not require all of the features of general-purpose knowledge representation platforms, as we discuss below.

The specification-linking rules (implemented as objects) in CATALOGEXPLORER use backward chaining to infer requirements. A rule interpreter such as JOSHUA [50] could be used for implementing the inference engine but the mechanism is simple enough that a specialized implementation based on CLOS methods turned out to be more straightforward and more efficient.

In EXPLAINER, most of the work involves searching different semantic networks comprising an example. Since EXPLAINER works with only one example at a time, the search space is relatively small (in general in the order of one to two thousand objects) making the time to find and present information negligible. For generating text, sufficiently good results have been obtained just by using patterns stored on the descriptors for different classes and perspectives. This implementation decision is a trade-off against using more sophisticated generation packages such as PENNMAN [33] or MUMBLE [34]. These systems are currently too slow and memory intensive to be used as components in interactive interfaces, their goal being more to support the careful analysis of language structures.

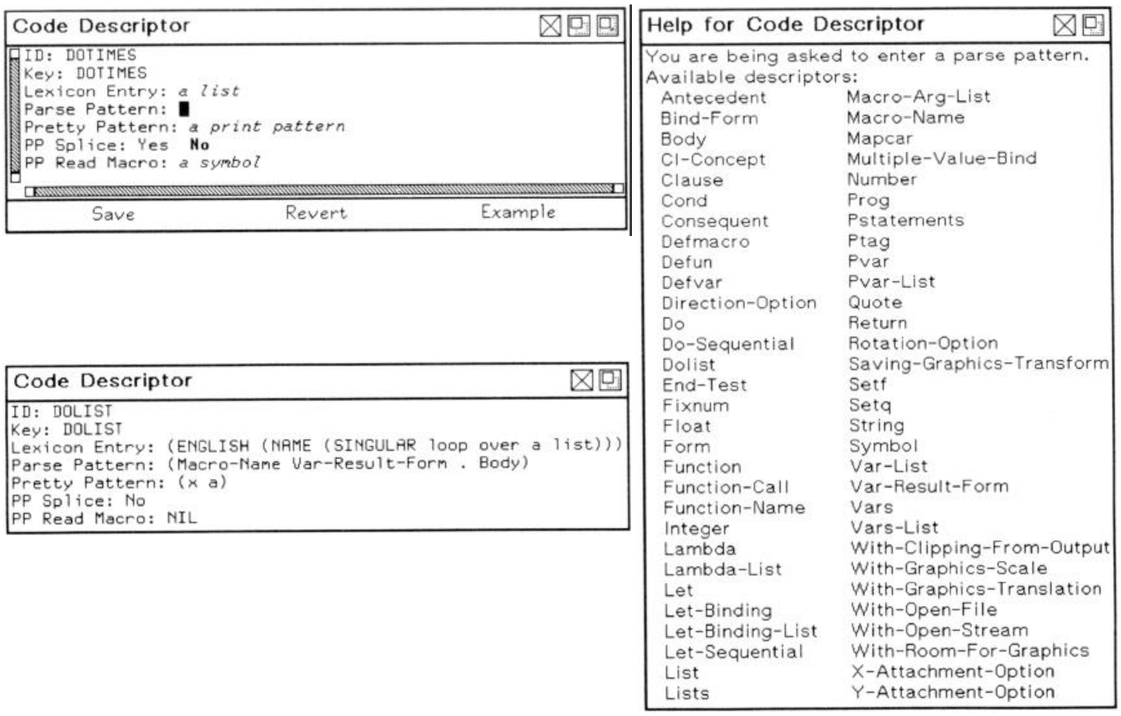
MODIFIER is supported by the SYMBOLICS presentation substrate, a package similar to the new CLIM standard [51]. Our extensions to provide editing of class objects through property sheets and to support subsumption for new classes have been reused in other design environment applications [22].

IV. LESSONS LEARNED

Evolution of knowledge-based design environments is crucial because it is not possible for environment builders to anticipate all the information that will be relevant in all future cases. Design information can never be complete, and design practices change with time [41], [42], [56]. Good concepts should be reused, and new concepts should be readily incorporated into the knowledge base as they arise. End users of the design environment should be able to modify tools such as we have presented without requiring knowledge about low-level programming. In this section, we first discuss various issues identified while evaluating our system-building efforts and then focus on the idea of evolution of knowledge and tools.

A. User Studies

Preliminary observations and user studies were performed with our three systems. These studies provided insight into improving the tools themselves as well as insights into the conceptual framework of retrieval, explanation, and end-user modifiability.



```
Code Descriptor
ID: DOTIMES
Key: DOTIMES
Lexicon Entry: a list
Parse Pattern: █
Pretty Pattern: a print pattern
PP Splice: Yes  No
PP Read Macro: a symbol
```

```
Code Descriptor
ID: DOLIST
Key: DOLIST
Lexicon Entry: (ENGLISH (NAME (SINGULAR loop over a list)))
Parse Pattern: (Macro-Name Var-Result-Form . Body)
Pretty Pattern: (x a)
PP Splice: No
PP Read Macro: NIL
```

```
Help for Code Descriptor
You are being asked to enter a parse pattern.
Available descriptors:
  Antecedent          Macro-Arg-List
  Bind-Form           Macro-Name
  Body                Mapcar
  Cl-Concept          Multiple-Value-Bind
  Clause              Number
  Cond                Prog
  Consequent          Pstatements
  Defmacro            Ptag
  Defun               Pvar
  Defvar              Pvar-List
  Direction-Option    Quote
  Do                  Return
  Do-Sequential       Rotation-Option
  Dolist              Saving-Graphics-Transform
  End-Test            Setf
  Fixnum              Setq
  Float               String
  Form                Symbol
  Function            Var-List
  Function-Call       Var-Result-Form
  Function-Name       Vars
  Integer             Vars-List
  Lambda              With-Clipping-From-Output
  Lambda-List         With-Graphics-Scale
  Let                 With-Graphics-Translation
  Let-Binding         With-Open-File
  Let-Binding-List    With-Open-Stream
  Let-Sequential      With-Room-For-Graphics
  List                X-Attachment-Option
  Lists               Y-Attachment-Option
```

Fig. 8. Modifying objects in EXPLAINER. In order to add knowledge about LISP construct, a new code descriptor that describes how LISP code can he parsed and printed has to he 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.

CATALOGEXPLORER: Preliminary observations for CATALOG EXPLORER focused on designers locating reusable objects in software development. The lack of appropriate support mechanisms for locating reusable objects is a primary obstacle hindering active software reuse. Designers need to access reusable objects relevant to the task at hand even when they are not able to articulate exactly what they need, or even know that potentially useful information exists. CATALOGEXPLORER circumvents these situations somewhat by inferring the task at hand from a partial specification.

While building CATALOGEXPLORER, we were concerned with whether designers would be frustrated by the possible incorrect identification of their task at hand and whether designers might become confused by the behavior of the system. In the preliminary studies, subjects wanted to know the rationale behind the system retrieving software objects and how the retrieved objects were related to their task at hand. Being derived from the domain argumentation base, the specification-linking rules can provide designers with casual relationships about the retrieval. While the method of computing appropriateness values that CATALOGEXPLORER currently uses is comparatively naive, the subjects appreciated its simplicity. The need for a mechanism that allows designers to modify an identified task (i.e., a constructed query) was identified.

If designers are not aware of the existence of potentially reusable information, they are not motivated to retrieve it. In these situations, the design environments should "deliver" information relevant to their current problem. Through the integration, the environments can partially identify the designer's intent, as well as detect occurrences of breakdowns in their partially constructed design solutions. When the environments become aware of such breakdowns, the systems will construct queries for designers and retrieve information relevant to the detected breakdowns. This notion of information delivery will complement the information access mechanisms and provide more effective information retrieval techniques for designers.

EXPLAINER: In planning improvements to the EXPLAINER system, we observed subjects solving simple graphic programming tasks independent of EXPLAINER in order to determine ideally what types of questions designers would ask about an example, and in general, what kinds of information they would seek. Subjects were given a programming task to complete and an example from which to start. They were told that the example was related to a potential solution of the programming task, as if a query and location had already taken place. They worked in an EMACS editor buffer and directed questions and comments to a human consultant. Subjects were observed through questions they asked, spontaneous talking aloud, and mouse cursor movements as they studied different parts of an example's code. Questions were restricted by the observer to questions about the example and not about the task.

Overall, subjects followed a prototyping behavior cycle when working with an example. They studied and asked questions to determine what parts of a program example were critical to their task. The subjects modified the critical sections and tested the new function they were building. On subsequent passes, subjects directed their attention to more detailed aspects of the example.

The perspective of the questions and relevance judgment focused on the LISP and plotting perspectives. For example, relevance of examples was initially judged by their visual or graphic features, such as whether they contained a circle or plot axis. Less experienced programmers asked questions about function parameters or programming constructs. Although the observer was prepared to answer questions about the domain of the plots, subjects did not request this information.

In summary, subjects working from examples followed a prototyping cycle, made use only of basic perspectives such as LISP and plotting, and took advantage of the visual nature of our domain. The issue of usefulness of domain perspectives is a question we plan to investigate further. We believe that the simplicity of examples and tasks tested so far contributed to the lack of a need for domain perspectives.

MODIFIER: Two user studies were carried out. In both, subjects were asked to complete different tasks in MODIFIER that required them to define new objects or to modify existing ones. The first study showed that help texts provided by MODIFIER about a modification task were insufficient. Furthermore, subjects had difficulty decomposing the modification tasks and making use of the information provided by the system. These findings led to new principles for explanations and showed the need for system-maintained task agendas. A task agenda was implemented as an advertisement mechanism, a method to draw the users' attention to the work materials that bear more work [57].

In the second study, the previous difficulties were alleviated. The advertisement mechanism guided the subjects smoothly through the task of defining a new class. The help texts emphasized examples of modification tasks in addition to textual descriptions. An "example button" retrieved classes and objects similar to the ones being modified. Although the retrieval mechanism for examples used a fairly simply matching algorithm (it used all slots in the new objects that already had values and looked for the object with the most matching values), the subjects found this button very useful. These experiments further contributed to our understanding of peoples' reliance on examples in completing tasks.

B. Maintenance—Enhancements by End Users

Design environments and design artifacts created in such environments are both software systems. Empirical analyses have shown that more than half of the effort in the development of complex software goes into maintenance. The data show that 75% of the maintenance effort goes into enhancements, such as changing systems to meet additional requirements [9]. In order to make maintenance a "first class citizen" activity in the lifetime of an artifact, the following are required.

- Design and development processes are more efficient if the reality of change is accepted explicitly. A lesson we learned in our work on end-user modifiability [16] is that there is no way to modify a system without detailed programming knowledge unless modifiability was an explicit goal in the original design of the system.
- The up-front costs (based not only on designing for what is desired and known at the moment but also anticipating changes and creating structures to support these changes) have to be acknowledged and dealt with.

The evolution of a software system from this perspective occurs as a feedback mechanism by responding to discrepancies between a system's actual and desired states. This allows one to adapt the system to changes without requiring these changes in detail at the original design time. The possibility for domain experts to change systems provides a potential solution to address the maintenance problems in software design. Users of a system are knowledgeable in the application domain and know best which enhancements are needed. An end-user modification component supports these users in adding enhancements to the system without the help of the system developers. End-user modifiable systems will take some of the burden to implement enhancements away from the system developer.

Not every desired enhancement can be implemented within the end-user modification component. In some cases, the system developers will have to change the system with more traditional means. It is also likely that the users do not use the "best" method for modifying the system. The system developers can collect the modifications done by the users from time to time and incorporate them into the next version of the system. Domain experts might not be able or willing to do major changes and reconceptualizations. Therefore, the evolutionary growth of our systems will be supplemented by major revisions performed by knowledge engineers.

C. Design for End-User Modifiability

Making a system end-user modifiable introduces additional costs during system development. However, Henderson and Kyng [25] argue that end-user modifiability is still advantageous because the resources saved in the initial development by ignoring end-user modifiability will be spent several times over during the system's lifetime. Several principles for making systems end-user modifiable have been identified during the development of MODIFIER and its integration and evaluation with the other systems implemented with our approach [16], [19], [22]. These principles, discussed below, include layered architectures, parameterization, explanations, task agendas, and critics.

Layered architectures have been used successfully in many areas, such as the design of operating systems [44]. They narrow the gap between the system space and the problem domain addressing the thin spread of application knowledge [10]. If a change extends beyond the functionality provided by one layer, users are not immediately thrown back to the system space but can descend one layer at a time. Our effort to develop domain-oriented environments provided an important step towards more user-accessible tailoring by creating high-level, or application-oriented, building blocks together with application-oriented ways of manipulating them.

Parameterization relieves the user of the task of modifying program code and locating the program parts responsible for a certain behavior. End users change only those parameters to change the behavior of the system, and can be supported in such modifications much better than in the modification of program code. These parameters have a meaning at higher layers of the layered architecture. The user can choose values for these parameters that the designer of the system did not foresee, but it is not possible to modify unparameterized parts of the system without resorting to programming.

The system must provide help about the possible values and the purpose of a parameter a user wants to modify. For example, during the modification of a program code descriptor in EXPLAINER, the software designer can ask the system to list all existing code descriptors that could be used in the parse pattern. In addition, an explanation has to be provided for code descriptors and what consequences would result from putting certain descriptors into the parse pattern. The form of explanation preferred during testing was explanation by example. Suchman's work on situated action [48] supports this observation, illustrating that people reason best in the context of a situation.

Decomposing modification tasks into manageable elements is an important part of a modification process. According to Jeffries *et al.* [26], novices are incapable of performing and recursively refining such task decomposition. Therefore, the system has to aid the user in decomposing the task by determining the relevant issues and directing the user's attention to them. To do so, the system needs a representation of the modification task. A task agenda such as the one used in DETENTE [57] can maintain the steps in a task and check their preconditions and states. DETENTE embeds agendas into application interfaces, maintains tasks, and advertises task recommendations. Checklists [31] are another method for guiding a user through a task.

Critics, although emphasized more in other systems of ours [15], play a critical role during a modification process. Critics are advisory systems that act as demons watching out for certain conditions, notifying users of such conditions, and proposing remedial actions. For instance, when adding a new descriptor in EXPLAINER (see Fig. 8), critics can signal that constraints between different fields are violated, e.g., that the parse and print pattern of a code descriptor do not match. Critics can also make users aware that the new descriptor is similar to an existing one, and suggest the two be merged.

V. RELATED WORK

Several software reuse systems maintain representations of what we have referred to as the higher-level specification or problem-domain knowledge. They use formal, automated techniques to produce new programs. The REQUIREMENT APPRENTICE [39] supports designers in the reuse of requirement specifications through cliches, commonly occurring structures that exist in most engineering disciplines. Cliches support designers in framing a problem. However, the approach is based on the waterfall model, and does not allow designers to intertwine problem specification and solution construction. DRACO [36] also stores requirement specifications, but it uses formal approaches to automatically derive designs from the specifications and does not reuse the specifications to help designers frame new problems.

LASSIE [11] and ARIES [23] overlap with our approach in their use of knowledge bases for supporting software reuse. 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' formulation of problem requirements. Our approach complements these aspects by stressing an integrated environment for designers that supports the concurrent development of requirements specifications and solution constructions, and the delivery of prestored objects related to the task at hand.

In common with our approach, domain analysis techniques [38] 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 [5]. 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.

Some software environments support designers with knowledge about software objects and development processes. MARVEL [24] uses rules for coordinating and integrating software development activities. An early tool, MASTERSCOPE [52] assisted users in "analyzing and cross-referencing" a LISP program, computing information about function calls and variable usage. While these tools can support the design process *per se*, they cannot ensure better solutions with respect to a problem domain.

In general, the distinguishing principle 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.

VI. CONCLUSIONS

Software design incorporates many cognitive activities, such as recognizing and framing a problem, understanding reusable information, and adapting information to a specific situation. Many researchers have turned their attention to knowledge-based and domain-oriented tools to support these processes. Our approach has been to combine such techniques into integrated domain-oriented design environments.

By integrating cooperative problem-solving approaches with knowledge-based techniques, we have developed a conceptual framework for supporting coevolution of problem specifications and software implementation that focuses on the role of human designers. This human-centered approach takes advantage of peoples' ability to understand and incrementally reformulate their problems, while allowing them to contribute to the gradual improvement of the underlying knowledge base. The notion of evolution circumvents the inability of the original builders of a design environment to anticipate all future needs and knowledge for complete coverage of a domain.

Within our conceptual framework, tools supporting location (CATALOGEXPLORER), comprehension (EXPLAINER), and modification (MODIFIER) of software objects have been implemented. Preliminary studies have shown these tools to offer promising solutions to providing design objects relevant to the task at hand, making design objects more comprehensible to designers, and anticipating potential changes at the hands of the end users. Admittedly, our approach is knowledge intensive. An important issue for our future work, and for other researchers in this area, is how efficient and durable the evolutionary process that we envision will be in large-scale projects.

ACKNOWLEDGMENT

The authors would like to thank all the members of the Human-Computer Communication Group at the University of Colorado who contributed to the development of our design environment approach.

REFERENCES

[1] L.B. Archer, "Whatever became of design methodology," in *Developments in Design Methodology, N. Cross,* Ed. New York: Wiley, 1984, pp. 347-349.

[2] D. Barstow, "A perspective on automatic programming." in *Proc. Eighth Int. Joint Conf. Artificial Intelligence, pp.* 1170-1179: 1983.

[3] D.R. Barstow. H.E. Shrobe, and E. Sandewall, Eds., *Interactive Programming Environments* New York: McGraw-Hill, 1984.

[4] F.L. Bauer, "Programming as an evolutionary process," in *Proc. Second Int. Conf. Software Engineering,* pp. 223-234, 1976.

[5] T.J. Biggerstaff, "Design recovery for maintenance and reuse," *IEEE Computer,* vol. 22, pp. 36-49, July 1989.

[6] D.G. Bobrow and T. Winograd, "An overview of KRL: A knowledge representation language," *Cognitive Science, vol.* 1, pp. 3-46, 1977.

[7] B.W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer,* vol. 21, pp. 61-72, May 1988.

[8] J. Conklin and M. Begeman, "gIBIS: A hypertext tool for exploratory policy discussion," *Trans. Office Information Systems,* vol. 6, pp. 303-331, Oct. 1988.

[9] Computer Science and Technology Board, "Scaling up: A research agenda for software engineering," *Commun. ACM,* vol. 33, pp. 281-293, Mar. 1990.

[10] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Commun. ACM,* vol. 31, pp. 1268-1287, Nov. 1988.

[11] P. Devanbu, R.J. Brachman, P.G. Sefridge, and B.W. Ballard, "LaSSIE: A knowledge-based software information system," *Commun. ACM,* vol. 34, pp. 34-49, 1991.

[12] D.C. Engelbart. "A conceptual framework for the augmentation of man's intellect," in *Computer-Supported Cooperative Work: A Book of Readings,* I. Greif, Ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 35-66, ch. 2.

[13] G. Fischer, T. Mastaglio, B.N. Reeves, and J. Rieman, "Minimalist explanations in knowledge-based systems," in *Proc. 23rd Hawaii Int. Conf. System Sciences,* vol. III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr, Ed., pp. 309-317, 1990.

[14] G. Fischer, A.C. Lemke, R. McCall, and A. Morch. "Making argumentation serve design," *Human Computer Interaction,* vol. 6, pp. 393-419, 1991.

[15] G. Fischer, A.C. Lemke, T. Mastaglio, and A. Morch, "The role of critiquing in cooperative problem solving," *ACM Trans. Inform. Syst.,* vol. 9, pp. 123-151, 1991.

[16] G. Fischer and A. Girgensohn, "End-user modifiability in design environments," in *Human Factors in Computing Systems. CHI'90 Conf. Proc.,* pp. 183-191, Apr. 1990.

[17] G. Fischer, S.R. Henninger, and D.F. Redmiles, "Cognitive tools for locating and comprehending software objects for reuse," in *Proc. Thirteenth Int. Conf. Software Engineering,* pp. 318-328, 1991.

[18] G. Fischer and A.C. Lemke, "Constrained design processes: Steps towards convivial computing," in *Cognitive Science and its Application for Human-Computer Interaction,* R. Guindon, Ed. Hillsdale, NJ: Lawrence Erlbaum, 1988, pp. l-58, ch. 1.

[19] G. Fischer, R. McCall, and A. Morch, "JANUS: Integrating hypertext with a knowledge-based design environment," in *Proc. Hypertext'89,* pp. 105-117, Nov. 1989.

[20] G. Fischer and K. Nakakoji, "Beyond the macho approach of artificial intelligence: Empower human designers—Do not replace them," *Knowledge-Based Systems J.,* to be published.

[21] G. Fischer and B.N. Reeves, "Beyond intelligent interfaces: Exploring, analyzing and creating success models of cooperative problem solving," *Applied Intelligence,* special issue, intelligent interfaces, to be published.

[22] A. Girgensohn and F. Shipman, "Supporting knowledge acquisition by end users: Tools and representations," in *Proc. Symp. Applied Computing,* pp. 310-348, Mar. 1992.

[23] D.R. Harris and W.L. Johnson, "Sharing and reuse of requirements knowledge," in *Proc. 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conf.,* pp. 65-77, Sept. 1991.

[24] G.T. Heineman, G.E. Kaiser, N.S. Barghouti. and I.Z. Ben-Shaul, "Rule chaining in MARVEL: Dynamic binding of parameters," in *Proc. 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conf.,* pp. 276-287, Sept. 1991.

[25] A. Henderson and M. Kyng, "There's no place like home: Continuing design in use," in *Design at Work: Cooperative Design of Computer Systems,* J. Greenbaum and M. Kyng, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates. 1991, pp. 219-240, ch. 11.

[26] R. Jeffries, A.A. Turner, P.G. Polson, and M. Atwood, "The processes involved in designing software," in *Cognitive Skills and Their Acquisition,* J.R. Anderson. Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1981, pp. 255-283, ch. 8.

[27] K. Kishida. T. Katayama, M. Matsuo, I. Miyamoto. K. Ochimizu, N. Saito, J.H. Sayler, K. Torii, and L.G. Williams: "SDA: A novel approach to software environment design and construction," in *Proc. 10th Int. Conf. Software Engineering,* pp. 69-79, Apr. 1988.

[28] J.L. Kolodner, "Improving human decision making through case-based decision aiding," *AI Magazine,* vol. 12, pp. 52-68, Summer 1991.

[29] W. Kunz and H.W.J. Rittel, "Issues as elements of information systems," Working Paper 131, Center for Planning and Development Research, Univ. California, 1970.

[30] J. Lee, "SIBYL: A tool for managing group decision rationale," in *Proc. Conf. Computer-Supported Cooperative Work,* pp. 79-92, Oct. 1990.

[31] A.C. Lemke and G. Fischer, "A cooperative problem solving system for user interface design," in *Proc. AAAI-90,* pp. 479-484, Aug. 1990.

[32] A. MacLean, R. Young, and T. Moran, "Design rationale: The argument behind the artifact." *in Proc. Conf. Human Factors in Computing Systems,* pp. 247-252, 1989.

[33] W.C. Mann. "An introduction to the Nigel text generation grammar," in *Systemic Perspectives on Discourse: Selected Theoretical Papers from the 9th International Systemic Workshop,* J.D. Benson, R.O. Freedle, W.S. Greaves, Eds., Ablex, 1985, pp. 84-95, ch. 4.

[34] M.W. Meteer, D.D. MacDonald, S.D. Anderson, D. Forster, L.S. Gay, A.K. Huettner, and P. Sibun, "Mumble-86: Design and implementation," *Coins Tech. Rep.* 87-87, Computer and Information Science, Univ. Massachusetts at Amherst, Sept. 1987.

[35] J. Moore and A. Newell, "How can MERLIN understand?," in *Knowledge and Cognition,* L.W. Gregg, Ed. Potomac, MD: Erlbaum, 1974, pp. 201-252.

[36] J. M. Neighbors, "The Draco approach to constructing software from reusable components," *IEEE Trans. Software Eng.,* vol. SE-10, pp. 564-574, Sept. 1984.

[37] J. Nielsen and J.T. Richards, "The experience of learning and using Smalltalk," *IEEE Software,* pp. 73-77, May 1989.

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

[39] H.B. Reubenstein, "Automated acquisition of evolving informal descriptions," *AI-TR 1205,* MIT, 1990.

[40] C.H. Rich and R.C. Waters, "Automatic programming: Myths and prospects," *Computer,* vol. 21, pp. 40-51, Aug. 1988.

[41] H.W.J. Rittel, "Second-generation design methods," in *Developments in Design Methodology,* N. Cross, Ed. New York: Wiley, 1984, pp. 317-327.

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

[43] E. Schoen, R.G. Smith, and B.G. Buchanan, "Design of knowledge-based systems with a knowledge-based assistant," *IEEE Trans. Software Eng.*, vol. SE-14, pp. 1771-1791, Dec. 1988.

[44] A. Silverschatz and J.L. Peterson, *Operating System Concepts.* Reading, MA: Addison-Wesley Publishing Company, 1988.

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

[46] S. Slade, "Case-based reasoning: A research paradigm," *AI Magazine*, vol. 12, pp. 42-55, Spring 1991.

[47] G.L. Steele, *Common LISP: The Language.* Burlington, MA: Digital Press, 1990, 2nd ed.

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

[49] W.R. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Commun. ACM,* vol. 25, pp. 438-439, July 1982.

[50] Symbolics, Inc., User's Guide to Basic Joshua, Cambridge, MA, 1988.

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

[52] W. Teitelman and L. Masinter, "The interlisp programming environment," in *Interactive Programming Environments,* D.R. Barstow, H.E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 83-96.

[53] J.H. Walker, "Document examiner: Delivery interface for hypertext documents," Hypertext'87 Papers, University of North Carolina, Chapel Hill, NC, pp. 307-323, Nov. 1987.

[54] R.C. Waters, "The programmer's apprentice: A session with KBEmacs," *IEEE Trans.* Software *Eng.*, vol. SE-11, pp. 1296-1320, Nov. 1985.

[55] D.A. White, "The knowledge-based software assistant: A program summary," in *Proc. 6th Annual Knowledge-Based Software Engineering Conf.,* pp. vi-xiii, Sept. 1991.

[56] T. Winograd and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design.* Norwood, NJ: Ablex, 1986.

[57] D.A. Wroblewski, T.P. McCandless, and W.C. Hill, "DETENTE: Practical support for practical action," in *Proc. Conf. Human Factors in Computing Systems,* pp. 195-202, 1991.

Gerhard Fischer is a professor in the Computer Science Department and a member of the Institute of Cognitive Science at the University of Colorado, Boulder.

His research interests include artificial intelligence, human-computer communication, cognitive science, and software design. His research has led to the development of new conceptual frameworks and to the design and implementation of a number of innovative systems in the areas of cooperative problem solving, and integrated domain-oriented design environments supporting software design.

Andreas Girgensohn received the M.S. degree in computer science from the University of Stuttgart, Germany, in 1987 and is currently a Ph.D. student in the Department of Computer Science and the Institute of Cognitive Science at the University of Colorado, Boulder.

His research interests include artificial intelligence, human-computer communication, cognitive science, and software design. His Ph.D. research (with Gerhard Fischer) focuses on supporting end-user modifiability in knowledge-based design environments.

Kumiyo Nakakoji received the B.A. degree in computer science from Osaka University, Japan, in 1986 and the MS. degree in computer science from the University of Colorado, Boulder, in 1990, and is currently a Ph.D. student in the Department of Computer Science and the Institute of Cognitive Science at the University of Colorado, Boulder.

Her studies are sponsored through a scholarship from Software Research Associates, Inc., Japan, where she has been an employee since 1986. Her Ph.D. research (with Gerhard Fischer) focuses on creating knowledge delivery mechanisms in design environments.

David Redmiles received the B.S. degree in mathematics and computer science in 1980 and the M.S. degree in computer science in 1982, both from the American University, Washington, D.C., and is currently a Ph.D. student in the Department of Computer Science and the Institute of Cognitive Science at the University of Colorado, Boulder.

His Ph.D. research (with Gerhard Fischer) focuses on the representation and reuse of catalog examples in design environments. Before coming to the University of Colorado, he worked in the Center for Computing and Applied Mathematics at the National Institute of Standards and Technology, Gaithersburg, MD.