Improving the Explanatory Power of Examples by a Multiple Perspectives Representation

Abstract

We developed a software tool called EXPLAINER for helping programmers complete new tasks by exploring previously worked-out examples. The implementation is based on the principle of making examples accessible through multiple perspectives and, specifically, perspectives that emphasize the programming plans underlying an example. The initial version of EXPLAINER used a simple, semantic network to represent multiple perspectives. A frame-based knowledge representation language called FrameTalk provides a more structured means of representing examples in EXPLAINER. Moreover, FrameTalk provides mechanisms that avoid deficiencies that arise when concept taxonomies must serve the dual purpose of representing specialization and composition of attributes.

Content Areas

Knowledge-based Support Systems / Representation / User Interfaces / Design

1 Introduction

The EXPLAINER tool was developed to help programmers solve tasks in computer graphics by allowing them to explore previously worked-out examples from several different perspectives. Perspectives are used by EXPLAINER to provide programmers access to information about an example through code listings, execution output, text statement of a problem domain, text statement of program features, and diagrams of program components and features.

The benefit of using examples in programming has intuitive appeal but has also been demonstrated in studies [PirolliAnderson 85, KesslerAnderson 86, Lewis 88]. These studies indicate that people who are able to develop good mental models of examples can apply that knowledge to new tasks. In programming, the *mental model* of a task has been defined as a person's informal understanding or background knowledge of a task and the knowledge or strategy to transform that informal understanding into a formal model, such as a program code [Fischer 87, KintschGreeno 85, Pennington 87].

The strategy for transforming a programming task into a solution is commonly termed a *programming plan* [Soloway et al. 88, SolowayEhrlich 84]. Programming plans are treated in analyses of programming as a single kind of knowledge. However, they

actually interrelate knowledge from many perspectives. Specifically, the transformation of task information into the solution space indicates a mapping between at least two perspectives, the problem domain of the task and the formalism of the solution (usually a programming language).

Thus, in the EXPLAINER tool illustrated below, a programming example is represented by the many perspectives and interrelationships that form the programming plan for the example. The translation of elements of a task into elements of a program solution is captured. Explanation consists of various techniques to visualize the different perspectives and interrelationships. In a sense, the representation of the example tries to capture the many perspectives of the mental model the author of the example had when originally working out the example task.

To date, EXPLAINER has relied upon a simple semantic network for representing examples. Compared to object-oriented languages (e.g. CLOS), this approach has the advantage that relationships between objects can be evaluated dynamically and, in particular, perspective relationships can be implemented outside of a strict inheritance hierarchy. However, this approach fails to make the salient concepts and relationships explicit and thereby hides the underlying principles on which explanations are based. A new, frame-based knowledge representation language called FrameTalk provides a more structured means of representing examples in EXPLAINER. Moreover, FrameTalk provides mechanisms that avoid deficiencies that arise when concept taxonomies must serve the dual purpose of representing specialization and composition of attributes.

The remainder of this paper is organized as follows. Section 2 provides a scenario to describe the EXPLAINER tool and its usage. Section 3 describes the current representation used in EXPLAINER while Section 4 explores a new implementation based on the FrameTalk language and provides a formal definition of perspectives. Section 5 discusses additional benefits of the FrameTalk language compared with standard object-oriented knowledge representation languages. Section 6 discusses related work.

2 Program Examples, Perspectives, and Explanation

Suppose a programmer has a graphics task, such as the Clock Task described in Figure 1. The programmer may retrieve an example similar to the task and use the EXPLAINER tool shown in Figure 2 as an aid to understanding the example. Retrieval of examples relevant to a task is not discussed in this paper; instead, the reader is referred to the literature, e.g., [Henninger 91, PrietoDiaz 91, Tou et al. 82].

The EXPLAINER interface (see Figure 2) is similar to a hypermedia tool. This implementation allows minimal information about the example to be initially presented. Programmers can then decide which specific features of the example they want to

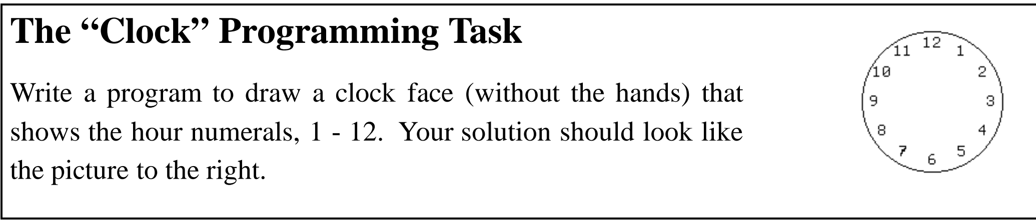


Figure 1: Clock Task as Described to Programmers

explore, presumedly choosing those most relevant to their current task. Information is accessed and expanded through a command menu (see Figure 3a). Almost all items presented on the screen are mouse sensitive and may be expanded.

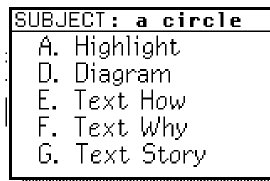
As explained above, the purpose of the EXPLAINER tool is to make apparent to programmers the programming plan underlying an example. This programming plan may have many interrelated perspectives. The example shown in Figure 2 solves the problem of visualizing addition modulo 100. Several perspectives are shown in this figure, including MODULO-ADDITION, CYCLIC-OPERATIONS, PLOT-FEATURES, PROGRAM-FEATURES, and LISP. Highlighting may be used to show the interrelationships between perspectives. For example, the highlighting between the perspectives enables the programmer to study the mapping between PLOT-FEATURES and LISP, i.e., between the labels around the circle and the LISP code to draw those labels.

Each perspective has a default presentation mode. Referring again to Figure 2, the LISP perspective is presented as a code listing (upper left pane), PLOT-FEATURES is presented in the sample execution (lower left) and in a diagram (upper right), CYCLIC-OPERATIONS and PROGRAM-FEATURES perspectives are described as text (lower right). Using the actions in the command menu shown in Figure 3a, a programmer may alter the default perspective presentation. For example, by selecting one of the "Text" commands from the menu, the programmer can have concepts in a LISP perspective presented as text descriptions instead of code. Conversely, each piece of information presented in the interface has a default perspective. The programmer may have information presented in an alternate perspective through the perspective menu shown in Figure 3b.

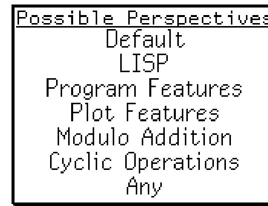
Thus, the EXPLAINER interface allows programmers to study the programming plan from different perspectives and to study the interrelationship of those perspectives. Figure 2 is the actual state of the EXPLAINER interface as left by a test subject. The subject was exploring the Modulo Addition Example to help program the Clock Task. The screen shows that the subject had

- expanded the diagram (upper right pane) to explore the components of labels in the PLOT-FEATURES perspective (by clicking on "more" cues);

Figure 2: Exploring the “Modulo Addition” Example in EXPLAINER



(a)



(b)

Figure 3: Pop-up Menus in EXPLAINER

- redrawn the initial diagram from its PLOT-FEATURES perspective to a LISP perspective (only a portion is visible in the middle of the upper right pane – menu action “Diagram”);
- retrieved a text description (lower right) about the concept of labels in the PROGRAM-FEATURES perspectives (menu action “Text Story”);
- generated a text description (lower right) of how labels are implemented in the LISP perspective (menu action “How”); and
- highlighted the concepts common across several different perspectives which pertain to labels (menu action “Highlight”)

This specific information enabled the test programmer to identify the LISP function called to draw the label, the assignment function that calculated the position, and what variables the position calculation depended on. The programmer could then apply the same problem decomposition, program structure, and 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.

Between the Modulo Addition Example and the Clock Task, the problem domain is plainly different. However, the two are similar with respect to plot features, such as circle drawing and labeling. This contrast of problem domains was selected deliberately in order to highlight the usefulness of multiple perspectives. Namely, multiple perspectives increase the chance of a program example matching a programming task.

The number and selection of perspectives for representing an example or any object is then clearly an issue. As with any knowledge-based application, the choices are dependent on the application domain. EXPLAINER was postulated as an aid to programmers accessing library functions, in this case, graphic functions. In this sense, it serves as a documentation substitute as opposed to a general-purpose reuse library or as an intelligent tutoring system. This choice clarifies the need for LISP

and PLOT-FEATURES perspectives and the mapping between the two. Commonality between problem domains would be serendipitous. In contrast, for EXPLAINER to serve as a general-purpose reuse library, scalability would be a major concern. To serve as an intelligent tutoring system, examples would not be chosen for coverage of software library features, but rather to show incrementally complex concepts in a domain. Which perspectives are needed depend on the nature of the domain and the range of context expected in tasks and subtasks within that domain.

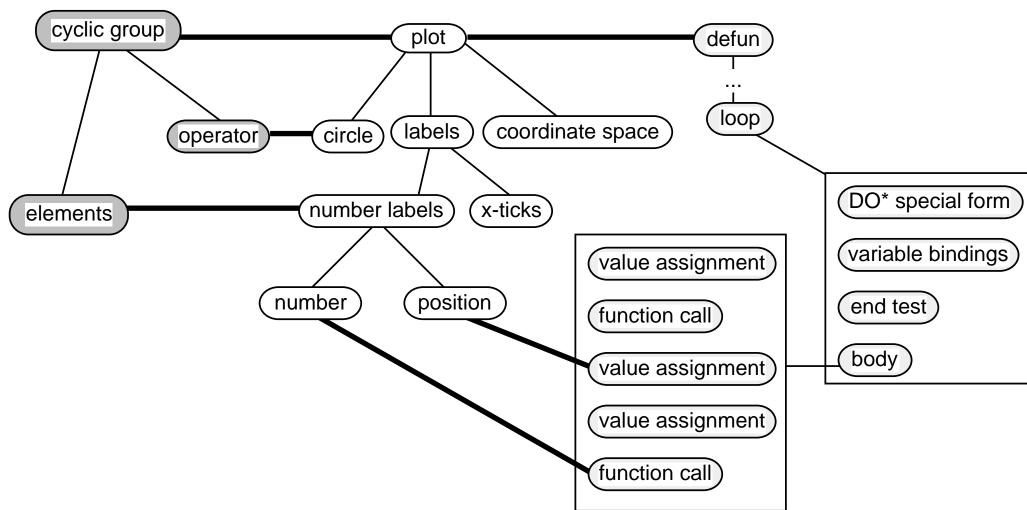
A study of 25 test programmers working from the Modulo Addition Example to create a program for the Clock Task, compared programmers using the EXPLAINER tool as a documentation aid versus a standard, though also hypermedia, function-oriented documentation aid. The study found that the programmers using EXPLAINER performed the programming task more directly and with less variability as a group. Programmers using the alternate aid proceeded in a trial-and-error fashion and, not surprisingly, exhibited great variability in their performance as a group. It is important to note that the reduction in variance was not at the cost of performance, i.e., good performers were about the same in both conditions. The reduction in performance variation resulted from "worse" programmers performing closer to the "better." Otherwise poor performers were helped by EXPLAINER.

3 A Semantic Network Approach to Representation

Originally, Explainer's knowledge was encoded in a semantic network, which was implemented using the Common Lisp Object System (CLOS). The nodes of the semantic network represented concepts in an example program code. For the Modulo Addition Example, the nodes instantiated concepts including the modulo operator, labels and a circle of the plot features, and function calls, arguments and value assignments of a LISP program (see Figure 4).

Instances of nodes are connected by three kinds of links: *components, roles* and *perspectives*. The component links connect one instantiation to zero or more instantiations which comprise it *in the same perspective*. In the above example, a PLOT consists of a COORDINATE-SPACE, a CIRCLE, and LABELS.

The component link captures the "how to" or implementation knowledge in an example. The role link is the inverse of the components link and supplies one kind of "why" or goal knowledge. For example, a "circle" is drawn as part of the graphics for a cyclic group. Instantiations exist in one specific perspective. However, they can have equivalent or analogous counterparts in other perspectives. For example the OPERATOR and ELEMENTS components of the CYCLIC-OPERATIONS perspective are equivalent to the CIRCLE and NUMBER-LABELS components of the PLOT-FEATURES perspective.



Three perspectives are shown in this figure (from left): CYCLIC-OPERATIONS (darkest ovals), PLOT-FEATURES (unshaded ovals), and LISP (shaded ovals). The thin lines represent components/roles links. The thick lines correspond to perspectives links.

Figure 4: Partial Representation of the Modulo Addition Example

In sum, instantiations in different perspectives are composed into hierarchies along the components/roles relation. The hierarchies in different perspectives are interrelated through the perspectives links of individual nodes. Thus, as a whole, an example program is one structured network that may be interpreted according to various perspectives.

The network is searched for neighboring nodes according to a parameterized search procedure. For instance, in the situation shown in Figure 2, the programmer applied the "Highlight" command (Figure 3a) to the LABELS concept presented in the diagram (upper right pane) or in the execution output (lower left). This action initiated a search through the network for instantiations which are related by perspective links to LABELS. Several related concepts, which may be traced in Figure 4, were identified and highlighted. At the least, any instantiation is related through the *root*, which is equivalenced to the main function of the LISP code for that example. Thus, any instantiation can be reached from any other, though some are distant with respect to the number of intervening links and nodes.

Thus, search begins with a concept that has been presented in the EXPLAINER interface. It traverses the links connecting nodes in the network and ends with a related node or nodes. The target is often immediate nodes on the components link to respond to a "how" request or the immediate nodes on the perspectives link to fulfill a highlighted request. However, the search can traverse both components and role links any number

of times, although cycles terminate the search in failure. The search pattern is basically breadth-first, though various characteristics of the path can be controlled to find the nodes in specific perspectives, or distances.

4 A Frame-based Approach to Representation

This semantic network approach to representation has some drawbacks which are related to semantic nets in general and specifically to the use of perspectives.

- The various relationships between nodes are not stated declaratively.
- The notion of perspectives is represented only indirectly by perspective links – perspectives do not have an identity in the net.
- The semantics of perspectives, components and roles links is "hidden" in the search procedure.

The notion of perspectives has been made an integral part of a frame-based representation formalism which replaces the semantic network approach. In FrameTalk, the perspectives mechanism was designed to provide additional structural support for representing knowledge about an application domain such as the one of the EXPLAINER system. The development of the perspectives mechanism for FrameTalk was guided by the following goals:

1. It should be possible to *define* concepts or terms through a composition of perspectives, e.g., a person has properties related to his physical existence, his family and social context, his working context, etc.
2. It should be possible to *identify* objects from different perspectives and to *apply* new perspectives to an object, e.g., depending on the context it should be possible to address a person as a student, employee, musician, etc.
3. There should be no structural difference between perspectives used for definition and perspectives used for identification, i.e., any perspective which is used for defining a concept must also be usable as an interpretation. The difference should be one of *usage* not one of structure.

Figure 5 provides an illustration of what perspectives mean in FrameTalk.

A perspective may be thought of as a set of properties for describing an object according to a particular, common theme. An object may be represented by several

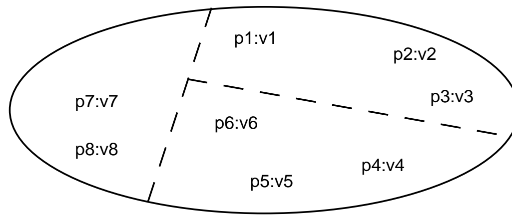


Figure 5: Perspectives of an Object

instantiations, each instantiation belonging to a distinct perspective and having specific values associated with that perspective's properties. For instance, the object in Figure 5 may be represented by any of the three sets: $\{p_1 : v_1, p_2 : v_2, p_3 : v_3\}$, $\{p_4 : v_4, p_5 : v_5, p_6 : v_6\}$ and $\{p_7 : v_7, p_8 : v_8\}$. Perspectives may be related, i.e., property-value pairs of one perspective may depend on property-value pairs of other perspectives. Thus, a perspective defines a set of properties for describing an object, and an instantiation of the perspective associates concrete values to the properties. By *applying* a perspective to an object, the corresponding property set (the instantiation) is *selected* as the representative for the object.

Figure 6 shows how the three perspectives for the Modulo Addition Example are specified.

```
(defframe modulo-addition ()
   (:perspectives
        (cyclic-operations (operator (:one function)(:diff components))
                           (elements (:all number)(:diff components)))
        (lisp (function (:one function)
                        (:diff components))
              (arguments (:all lisp-variable) (:diff components))
              (body (:all lisp-form) (:diff components)))
        (plot-features (coordinate-space (:diff components))
                       (circle (:diff components))
                       (labels (:all graph-label) (:diff components))))
   (:relations
        (same-object (cyclic-operations operator) (graphics circle))
        (same-object (cyclic-operations elements)
                     (plot-features labels number-labels))
```

Figure 6: FrameTalk Definition of the Modulo Addition Example

The Modulo Addition Example is represented by a frame consisting of three perspectives: CYCLIC-OPERATIONS, LISP and PLOT-FEATURES. Each of the perspectives consists of slots with slot descriptions, which restrict possible slot fillers. For instance, the OPERATOR slot of the CYCLIC-OPERATIONS perspective must be filled with

exactly one function and all of the values of the ELEMENTS slot must be numbers. In this example, all slots mentioned *differentiate* a COMPONENTS slot, which is defined at a more general level (not shown) in each of the three perspectives. Fillers of a component slot belong to the same perspective as the component slot.

The RELATIONS part of the frame specifies the *same-object* relation for components in different perspectives. This relation ensures that slot fillers point to the same object (though potentially in different perspectives).

The frame definition of Figure 6 is an abstract description of the Modulo Addition Example. The network of Figure 4 is subsumed by this abstraction. Some of the components/roles links and some of the perspectives links are specific for this example. All of them have been explicitly asserted by the programmer, for instance, the perspectives links between the position of the number labels in the PLOT-FEATURES perspective and the value assignment form in the the LISP perspective. They explicitly state that the LISP form corresponds to the positioning of the PLOT-FEATURES elements.

The definition is used to enforce perspective relations on instantiations. Asserting the fact that a given LISP program is described by the LISP perspective of the modulo addition frame generates instantiations for the other perspectives and establishes the necessary relationships. The remaining details, which are specific to the example, would be filled in by the programmer. The declarative form of the definition also more readily enables reuse of the knowledge from one example frame to the next.

5 Benefits of the Perspectives Mechanism

From a formal point of view, frames are composed of partial descriptions each of which belongs to a distinct perspective. Due to the perspectives structure of frames, an object is represented as a composition of instantiated partial descriptions. An object may be accessed by any of its instantiations, i.e., through any of its perspectives. Reasoning components of FrameTalk make use of this partitioning. For instance, the *realizer,* a component which associates partial descriptions to instantiations, does not cross a perspective's boundary, thereby significantly reducing its computational complexity.

The introduction of a perspectives mechanism into a frame language provides additional expressive power for taxonomic knowledge bases. Traditionally, a taxonomy of concepts is used for two separate purposes: specialization and combination of attributes.

For instance, concepts such as ELEMENTS, NUMBER-LABELS, and LOOP are combined for example into ELEMENTS-NUMBER-LABELS-LOOP in order to describe a concept fitting the common description of its possible uses. The new concept is made a

subconcept of the existing ones and thereby inherits all what is specific to them. This leads to some unwanted consequences for an application taxonomy:

- There is a combinatorial proliferation of concepts.
- Concepts tend to become artificial; they do not have a natural counterpart in the conceptual (and lexical) structure of the application domain.
- The taxonomic relationships between concepts do not allow a distinction between different dimensions of generalization. LOOP is a more general concept than ELEMENTS-NUMBER-LABELS-LOOP with respect to some plot feature. LABELS is more general than ELEMENTS-NUMBER-LABELS-LOOP with respect to properties which characterize LISP language components. The network of concepts represents multiple taxonomies which can not be separated based on its structure.

A related problem manifests itself when the generalization hierarchy with multiple super concepts is (mis-)used to combine partial definitions. In the area of object-oriented programming, these partial definitions are known as *mixin classes*. A super class link to a mixin class does not express generalization but composition of structure and behavior.

The need to compose descriptions from various sources is often caused by multiple usage contexts. In different application contexts, concepts are *used* in different ways. In the EXPLAINER domain, a “Modulo Addition Example viewed as a graphic plot” looks different than a “Modulo Addition Example viewed as a LISP code”. These differences in use are not expressed by the concept taxonomy. A perspectives mechanism provides the adequate means to account for the differences.

6 Related Work

Many knowledge-based systems exist or are being developed to support programmers and software designers. In general the distinctions could be summarized according to where emphasis is placed on a man-machine spectrum. Our emphasis while developing a knowledge-based tool has been on the design process as tightly coupled with a human designer and not on automation. A balance is sought between the creative abilities of the human and the ability to retrieve plans and examples by the computer. The assumption is that good design is still the domain of people, though systems can provide support [Brooks Jr. 87]. Systems supporting human designers have often focussed on only one perspective, the system/programming language perspective [RichWaters 90, TeitelmanMasinter 84]. Some systems support

decomposition from domain perspectives, though the knowledge is often organized around a domain taxonomy instead of human-centered needs [Devanbu et al. 91, HarrisJohnson 91].

In the introduction, it was noted that the example-based approach was in part based on a comprehension oriented theory in problem solving explored by Kintsch and Greeno [KintschGreeno 85, DijkKintsch 83]. Their original work was in the area of word arithmetic problems and later was extended with Mannes to the domain of simulating computing tasks [MannesKintsch 91]. Their approach is to simulate *how* people use *what kinds of* knowledge in performing these tasks. Problem solving interpreted in a comprehension framework is particularly well-suited for understanding the role of example-based support as there is a dual role for comprehension: comprehension of the problem or task, and comprehension of a supplied example related to the task. In adapting parts of their framework, the emphasis had to be added with respect to how examples would have to be explored by people from the different perspectives specifically involved in programming plans. The different perspectives additionally provide a finer-grained analysis than their two-part separation of a person's understanding into situation and problem (mental) models.

Aside from systems in the software domain, perspectives have played a role in explanation in expert systems [Swartout 83, BatemanParis 89] and intelligent tutoring systems [Souther et al. 89]. The role of perspectives in explaining results of expert system deductions has focused on providing alternative texts depending on the level of sophistication of the user and not, as in EXPLAINER, on the need to interpret the same information for different purposes. For example, in EXPLAINER, the Modulo Arithmetic Example illustrates both how to label a graph and how to program a loop. Similarly, in intelligent tutoring systems "perspectives" do not offer different interpretations of an example but rather different relationships between domains (e.g. "part-of," "dependent-on," "affected-by"). Furthermore, intelligent tutoring systems must represent a greater amount of knowledge than more focussed support systems such as EXPLAINER.

The notion of perspectives has been used in the areas of knowledge representation and object-oriented programming. In an early system called MERLIN [MooreNewell 73], perspectives are mappings between so-called β-structures. There, a perspective is an alternative way of expressing the same thing. [+ 3 5], [8], [ARITH-OP BINARY COMMUTATIVE [FUNCTION ADDITION] [FIRST-ARG 3] [SECOND-ARG 5]] are alternative descriptions for the same object.

Minsky [Minsky 75] introduced the notion of "frame-systems" for explaining a human's efficiency when interpreting visual scenes. Although movements around an object such as a cube changes the perceived image, there is no need to re-interpreted the entire image because multiple views are represented by frames which share information in common "terminals". The idea of sharing parts is extended to apply to

different frame-systems, i.e., collections of frames describing structurally different objects. As an example, an electricity generator can be described as a mechanical and as an electrical system. Although the term "perspective" is not explicitly used, the resemblance to our approach is obvious.

These notions of perspectives differ from the approach taken by KRL [BobrowWinograd 77]. KRL descriptions are partial characterizations which may be applicable to an object. Descriptions can be compared to determine whether they are compatible, i.e. whether they can describe the same object. In this sense, descriptions may be used to express different point of views. Although one of the descriptor types of KRL is named "perspective" it is not directly related to multiple views. It is used to relate a description to a prototype which resembles the instance-class relationship of an object-oriented programming language.

In the PIE system [BobrowGoldstein 80, GoldsteinBobrow 80a], perspectives were used to introduce multiple inheritance into SMALLTALK [GoldsteinBobrow 80b]. A distinct class NODE defined an instance variable PERSPECTIVES which stored each node's list of perspectives. As in the later LOOPS system [StefikBobrow 86], perspectives represented different aspects of an object and could be added and deleted dynamically. Instances of different classes comprised the perspectives of a single object. The sets of properties corresponding to each instance were distinct; there was no facility for specifying perspectives on the level of LOOPS concepts.

PIE and LOOPS are similar to our approach with respect to representing information about an object by several instantiations. They are different with respect to the motivation: Both systems extend an object-oriented programming language; we see the notion of perspectives as an improvement to the structuring of knowledge and as an augmentation to frame-based knowledge representation languages.

The TROPES system [Mariño et al. 90] is an object-oriented representation system which uses the notion of multiple but distinct inheritance hierarchies of classes to define a "concept". A single inheritance hierarchy constitutes a perspective. Instances are classified with respect to the perspectives-trees. Again, an object-oriented system is extended to allow the formulation of perspectives.

7 Conclusions

The proposed perspectives mechanism extends representation languages by providing additional structure for formulating and using knowledge in a taxonomy of concepts:

- Perspectives allow the formulation of *partial definitions*. Composition of properties can be structurally separated from specialization of properties.

- Different views can be expressed by combining properties in different ways. Objects may be described from different viewpoints by multiple perspectives.
- The perspectives structure can be used for comparing objects and answering questions for explanation purposes.

Furthermore, the multiple perspectives representation serves the dual role of providing a framework for organizing knowledge according to the needs of the mental models people rely upon during problem solving. This supports better provision and structuring of the knowledge represented for explanation.

The duality of perspectives for representation and perspectives for modeling the needs of human users is an elegant combination for a human-computer interface. It suggests an issue for the future, i.e., what properties a knowledge representation scheme for a human-computer application should have in order to promote the most natural balance and operation between computer representation of the knowledge and the human use of that knowledge. We believe this paper points to some of the desired characteristics.

References

[BatemanParis 89] J. Bateman and C. Paris. Phrasing a Text in Terms the User Can Understand. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (Detroit, MI)*, pp. 1511–1517, San Mateo, CA, August 1989. Morgan Kaufmann Publishers.

[BobrowGoldstein 80] D. Bobrow and I. Goldstein. *Representing Design Alternatives*. AISB, Amsterdam, 1980. Proceedings of the AISB Conference.

[BobrowWinograd 77] D. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

[Brooks Jr. 87] F. Brooks Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[Devanbu et al. 91] P. Devanbu, R. Brachman, P. Sefridge and B. Ballard. LaSSIE: A Knowledge-Based Software Information System. *Communications of the ACM*, 34(5):34–49, 1991.

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

[Fischer 87] G. Fischer. Cognitive View of Reuse and Redesign. *IEEE Software, Special Issue on Reusability*, 4(4):60–72, July 1987.

[GoldsteinBobrow 80a] I. Goldstein and D. Bobrow. Descriptions for a Programming Environment. In *Proceedings of AAAI*, pp. 187–189. Stanford, 1980.

[GoldsteinBobrow 80b] I. P. Goldstein and D. G. Bobrow. Extending Object Oriented Programming in Smalltalk. In *Proceedings of the First Lisp Conference*, Stanford University, 1980.

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

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

[KesslerAnderson 86] C. Kessler and J. Anderson. Learning Flow of Control: Recursive and Iterative Procedures. *Human-Computer Interaction*, 2:135–166, 1986.

[KintschGreeno 85] W. Kintsch and J. Greeno. Understanding and Solving Word Arithmetic Problems. *Psychological Review*, 92:109–129, 1985.

[Lewis 88] C. Lewis. Why and How to Learn Why: Analysis-Based Generalization of Procedures. *Cognitive Science*, 12(2):211–256, 1988.

[MannesKintsch 91] S. Mannes and W. Kintsch. Routine Computing Tasks: Planning as Understanding. *Cognitive Science*, 3(15):305–342, 1991. also published as Technical Report No. 89-8, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[Mariño et al. 90] O. Mariño, F. Rechenmann and P. Uvietta. Multiple Perspectives and Classification Mechanism in Object-Oriented Representation. In *Proceedings of the ECAI'90*, pp. 425–430, 1990.

[Minsky 75] M. Minsky. A Framework for Representing Knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*, pp. 211–277. McGraw Hill, New York, 1975.

[MooreNewell 73] J. Moore and A. Newell. How can MERLIN understand? In L. Gregg (Ed.), *Knowledge and Cognition*, pp. 201–310. Lawrence Erlbaum Associates, Hillsdale, NJ, 1973.

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

[PirolliAnderson 85] P. Pirolli and J. Anderson. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology*, 39(2):240–272, 1985.

[PrietoDiaz 91] R. Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 35(5), May 1991.

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

[Soloway et al. 88] E. Soloway, J. Pinto, S. Letovsky, D. Littman and R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[SolowayEhrlich 84] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[Souther et al. 89] A. Souther, L. Acker, J. Lester and B. Porter. Using View Types to Generate Explanations in Intelligent Tutoring Systems. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (Ann Arbor, MI)*, pp. 123–130, Hillsdale, NJ, August 1989. Lawrence Erlbaum Associates.

[StefikBobrow 86] M. Stefik and D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[Swartout 83] W. Swartout. XPLAIN: A System for Creating and Explaining Expert Consulting Programs. *Artificial Intelligence*, 21:285–325, 1983.

[TeitelmanMasinter 84] W. Teitelman and L. Masinter. *The Interlisp Programming Environment*, pp. 83–96. McGraw-Hill Book Company, New York, 1984.

[Tou et al. 82] F. Tou, M. Williams, R. Fikes, A. Henderson and T. Malone. RABBIT: An Intelligent Database Assistant. In *Proceedings of AAAI-82, Second National Conference on Artificial Intelligence (Pittsburgh, PA)*, pp. 314–318, August 1982.