Palo Alto Research Center

An Experimental Description-Based Programming Environment: Four Reports

By Ira Goldstein and Daniel Bobrow

XEROX

An Experimental Description-Based Programming Environment: Four Reports

by Ira Goldstein and Daniel Bobrow

CSL-81-3 March 1981

Abstract: See next page.

CR Categories: 4.22, 4.43, 4.32

Key words and phrases: Programming Environments, Man-Machine Systems, Object Oriented Program, Knowledge Representation.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Abstract:

This document reprints four articles that describe PIE, an experimental personal information environment, from the vantage point of its application to software development. PIE employs a description language to support the interactive development of programs. PIE contains a network of nodes, each of which can be assigned several perspectives. Each perspective describes a different aspect of the program structure represented by the node, and provides specialized actions from that point of view. Contracts can be created that monitor nodes describing different parts of a program's description. Contractual agreements are expressible as formal constraints, or, to make the system failsoft, as English text interpretable by the user. Contexts and layers are used to represent alternative designs for programs described in the network. The layered network database also facilitates cooperative program design by a group, and coordinated, structured documentation.

The first article, "Descriptions for a Programming Environment," provides an overview of PIE. The second article, "Extending Object Oriented Programming in Smalltalk," explores the generalizations made to the Smalltalk language in order to combine its strengths as an object language with capabilities usually found in AI description languages. This extended dialect is used to implement the PIE system. The third article, "Representing Design Alternatives," describes PIE's machinery for representing the evolution of a software design. This machinery is described in greater detail in a separate report, CSL-80-5. The fourth article, "Browsing in a Programming Environment," describes the user interface.

PIE has also been employed to represent office related information such as mail, calendars, documents, lectures and expense reports. These capabilities will be described in CSL-81-4, PIE: An Experimental Personal Information Environment, to be published later in 1981.

Table of Contents Page

These papers originally appeared as:

Goldstein, I. P. and Bobrow, D. G., "Descriptions for a Programming Environment," *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*, Stanford, Ca, August 1980.

Goldstein, I. P. and Bobrow, D. G., "Extending Object Oriented Programming in Smalltalk," *Proceedings of the Lisp Conference*, Stanford, Ca, August 1980.

Bobrow, D. G. and Goldstein, I. P., "Representing Design Alternatives, "*Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*," Amsterdam, July 1980.

Goldstein, I. P. and Bobrow, D. G., "Browsing in a Programming Environment" *(submitted for publication)*.

DESCRIPTIONS FOR A PROGRAMMING ENVIRONMENT[1]

1. Introduction

In most programming environments, there is support for the text editing of program specifications, and support for building the program in bits and pieces. However, there is usually no way of linking these interrelated descriptions into a single integrated structure. The English descriptions of the program, its rationale, general structure, and tradeoffs are second class citizens at best, kept in separate files, on scraps of paper next to the terminal, or, for a while, in the back of the implementor's head.

Furthermore, as the software evolves, there is no way of noting the history of changes, except in some primitive fashion, such as the history list of Interlisp [Teitelman78]. A history list provides little support for recording the purpose of a change other than supplying a comment. But such comments are inadequate to describe the rationale for coordinated sets of changes that are part of some overall plan for modifying a system. Yet recording such rationales is necessary if a programmer is to be able to come to a system and understand the basis for its present form.

Developing programs involves the exploration of alternative designs. But most programming environments provide little support for switching between alternative designs or comparing their similarities and differences. They do not allow alternative definitions of procedures and data structures to exist simultaneously in the programming environment; nor do they provide a representation for the evolution of a particular set of definitions across time.

In this paper we argue that by making descriptions first class objects in a programming environment, one can make life easier for the programmer through the life cycle of a piece of software. Our argument is based on our experience with PIE, a description-based programming environment that supports the design, development, and documentation of Smalltalk programs.

2. Networks

The PIE environment is based on a network of nodes which describe different types of entities. We believe such networks provide a better basis for describing systems than files. Nodes provide a uniform way of describing entities of many sizes, from small pieces such as a single procedure to much larger conceptual entities. In our programming environment, nodes are used to describe code in individual methods, classes, categories of classes, and configurations of the system to do a particular job. Sharing structures between configurations is made natural and efficient by sharing regions of the network.

1 Published in the Proceedings of the First Annual Conference of the American Association for Artificial Intelligence, August 1980, pp. 187-194.

Nodes are also used to describe the specifications for different parts of the system. The programmer and designer work in the same environment, and the network links elements of the program to elements of the design and specification. The documentation on how to use the system is embedded in the network also. Using the network allows multiple views of the documentation. For example, a primer and a reference manual can share many of the same nodes while using different organizations suited to their different purposes.

In applying networks to the description of software, we are following a tradition of employing semantic networks for knowledge representation. Nodes in our network have the usual characteristics that we have come to expect in a representation language--for example, defaults, constraints, multiple perspectives, and context-sensitive value assignments.

There is one respect in which the representation machinery developed in PIE is novel: it is implemented in an object-oriented language. Most representation research has been done in Lisp. Two advantages derive from this change of soil. The first is that there is a smaller gap between the primitives of the representation language and the primitives of the implementation language. Objects are closer to nodes (frames, units) than lists. This simplifies the implementation and gains some advantages in space and time costs. The second is that the goal of representing software is simplified. Software is built of objects whose resemblance to frames makes them natural to describe in a frame-based knowledge representation.

3. Perspectives

Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. For example, one view of a Smalltalk class provides a definition of the structure of each instance, specifying the fields it must contain; another describes a hierarchical organization of the methods of the class; a third specifes various external methods called from the class; a fourth contains user documentation of the behavior of the class.

The attribute names of each perspective are local to the perspective. Originally, this was not the case. Perspectives accessed a common pool of attributes attached to the node. However, this conflicted with an important property that design environments should have, namely, that different agents can create perspectives independently. Since one agent cannot know the names chosen by another, we were led to make the name space of each perspective on a node independent.

Perspectives may provide partial views which are not necessarily independent. For example, the organization perspective that categorizes the methods of a class and the documentation perspective that describes the public messages of a class are interdependent. Attached procedures are used to maintain consistency between such perspectives.

Each perspective supplies a set of specialized actions appropriate to its point of view. For example, the *print* action of the structure perspective of a class knows how to prettyprint its fields and class variables, whereas the organization perspective knows how to prettyprint the methods of the class. These actions are implemented directly through messages understood by the Smalltalk classes defining the perspective.

Messages understood by perspectives represent one of the advantages obtained from developing a knowledge representation language within an object-oriented environment. In most knowledge representation languages, procedures can be attached to attributes. Messages constitute a generalization: they are attached to the perspective as a whole. Furthermore, the machinery of the object language allows these messages to be defined locally for the perspective. Lisp would insist on global functions names.

4. Contexts and Layers

All values of attributes of a perspective are relative to a *context*. Context as we use the term derives from Conniver [SussmanMcDermott72]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is natural to create alternative contexts in which different values are stored for attributes in a number of nodes. The user can then examine these alternative designs, or compare them without leaving the design environment. Since there is an explicit model of the differences between contexts, PIE can highlight differences between designs. PIE also provides tools for the user to choose or create appropriate values for merging two designs.

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer. Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context the first time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Layers and contexts are themselves nodes in the network. Describing layers in the network allows the user to build a description of the rationale for the set of coordinated changes stored in the layer in the same fashion as he builds descriptions for any other node in the network. Contexts provide a way of grouping the incremental changes, and describing the rationale for the group as a whole. Describing contexts in the network also allows the layers of a context to themselves be asserted in a context sensitive fashion (since all descriptions in the network are context-sensitive). As a result, super-contexts can be created that act as *big switches* for altering designs by altering the layers of many sub-contexts.

5. Contracts and Constraints

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ contracts between nodes to describe these dependencies. Implementing contracts raises issues involving 1) the knowledge of which elements are dependent; 2) the way of specifying the agreement; 3) the method of enforcement of the agreement; 4) the time when the agreement is to be enforced.

PIE provides a number of different mechanisms for expressing and implementing contracts. At the implementation level, the user can attach a procedure to any attribute of a perspective, (see BobrowWinograd77 for a fuller discussion of attached procedures); this allows change of one attribute to update corresponding values of others. At a higher level, one can write simple constraints in the description language (e.g. two attributes should always have identical values), specifying the dependent attributes. The system creates attached procedures that maintain the constraint.

There are constraints and contracts which cannot now be expressed in any formal language. Hence, we want to be able to express that a set of participants are interdependent, but not be required to give a formal predicate specifying the contract. PIE allows us to do this. Attached procedures are created for such contracts that notify the user if any of the participants change, but which do not take any action on their own to maintain consistency. Text can be attached to such informal contracts that is displayed to the user when the contract is triggered. This provides a useful inter-programmer means of communication and preserves a *failsoft* quality of the environment when formal descriptions are not available.

Ordinarily such non-formal contracts would be of little interest in artificial intelligence. They are, after all, outside the comprehension of a reasoning program. However, our thrust has been to build towards an artificially intelligent system through successive stages of man-machine symbiosis. This approach has the advantage that it allows us to observe human reasoning in the controlled setting of interacting with the system. Furthermore, it allows us to investigate a direction generally not taken in AI applications: namely the design of memory-support rather than reasoning-support systems.

An issue in contract maintenance is deciding when to allow a contract to interrupt the user or to propagate consistency modifications. We use the closure of a layer as the time when contracts are checked. The notion is that a layer is intended to contain a set of consistent values. While the user is working within a layer, the system is generally in an inconsistent state. Closing a layer is an operation that declares that the layer is complete. After contracts are checked, a closed layer is immutable. Subsequent changes must be made in new layers appended to the appropraiate contexts.

6. Coordinating designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment in which potentially

overlapping partial designs can be examined without overwriting one another. This is accomplished by the convention that different designers place their contributions in separate layers. Thus, where an overlap occurred, the divergent values for some common attributes are in distinct layers.

Merging two designs is accomplished by creating a new layer into which are placed the desired values for attributes as selected from two or more competing contexts. For complex designs, the merge process is, of course, non-trivial. We do not, and indeed cannot, claim that PIE eliminates this complexity. What it does provides is a more finely grained descriptive structure than files in which to manipulate the pieces of the design. Layers created by a merger have associated descriptions in the network specifying the contexts participating in the merger and the basis for the merger.

7. Meta-description

Nodes can be assigned meta-nodes whose purpose is to describe defaults, constraints, and other information about their object node. Information in the meta-node is used to resolve ambiguities when a command is sent to a node having multiple perspectives.

One situation in which ambiguity frequently arises is when the PIE interface is employed by a user to browse through the network. When the user selects a node for inspection, the interface examines the meta-node to determine which information should be automatically displayed for the user. By appropriate use of meta-information, we have made the default display of the PIE browser identical to one used in Smalltalk. (Smalltalk code is organized into a simple four-level heirarchy, and the Smalltalk browser allows examination and modification of Smalltalk code using this taxonomy.) As a result, a novice PIE user finds the environment similar to the standard Smalltalk programming environment which he has already learned.

Simplifying the presentation and manipulation of the layered network underlying the PIE environment remains an important research goal, if the programming environment supported by PIE is to be useful as well as powerful. We have found use of a meta-level of descriptions to guide the presentation of the network to be a powerful device to achieve this utility.

8. Conclusion

PIE has been used to describe itself, and to aid in its own development. Specialized perspectives have been developed to aid in the description of Smalltalk code, and for PIE perspectives themselves. On-line documentation is integrated into the descriptive network. The implementors find this network-based approach to developing and documenting programs superior to the present Smalltalk programming environment. A small number of other people have begun to use the system.

This paper presents only a sketch of PIE from a single perspective. The PIE description language is the result of transplanting the ideas of KRL [BobrowWinograd77] and FRL [GoldsteinRoberts77] into the object oriented programming environment of Smalltalk [KayGoldberg77, Ingalls78]. A more extensive discussion of the system in terms of the design process can be found in BobrowGoldstein80, and GoldsteinBobrow80a. A view of the PIE

description language as an extension of the object oriented programming metaphor can be found in GoldsteinBobrow80b. Finally, the use of PIE as a prototype office information system is described in Goldstein80.

References

1. Bobrow, D.G. and Goldstein, I.P. "Representing Design Alternatives", *Proceedings of the AISB Conference*, Amsterdam, 1980
2. Bobrow, D.G. and Winograd, T. "An overview of KRL, A Knowledge Representation Language", *Cognitive Science* 1, 1 1977
3. Goldstein, I.P. "PIE: A Network-Based Personal Information Environment", *Proceedings of the Office Semantics Workshop*, Chatham, Mass., June, 1980.
4. Goldstein, I.P. and Bobrow, D.G., "A Layered Approach to Software Design", *Xerox Palo Alto Research Center CSL-80-5.* 1980a
5. Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference.* Stanford University, 1980b
6. Goldstein, I.P. and Roberts, R.B. "NUDGE, A Knowledge-Based Scheduling Program", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge: 1977, 257-263.
7. Ingalls, Daniel H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp 9-16.
8. Kay, A. and Goldberg, A. "Personal Dynamic Media" *IEEE Computer*, March, 1977.
9. Sussman, G., & McDermott, D. "From PLANNER to CONNIVER -- A genetic approach". *Fall Joint Computer Conference.* Montvale, N. J.: AFIPS Press, 1972.
10. Teitelman, W., *The Interlisp Manual*, Xerox Palo Alto Research Center. 1978

EXTENDING OBJECT ORIENTED PROGRAMMING IN SMALLTALK[1]

Object oriented programming is a powerful computational framework for many applications, and Smalltalk [Kay72] is a good example of a language that embodies this framework. Smalltalk is especially excellent for simulation, as one would expect from the fact that Simula [Dahl66] is part of its intellectual genealogy. Objects can represent the participants in a simulation; messages can represent their interactions. However, the 1976 implementation of Smalltalk [Ingalls76] lacks a number of capabilities that we believe can extend its power considerably, especially for applications (including simulation) that occur in the context of an overall design process. These capabilities arise from the assignment of different kinds of description to objects.

(1) *multiple perspectives*: the assignment of more than one point of view that allows inheritance of behavior from independent superclasses.

(2) *metadescription*: the assignment of constraints to attributes that allows the system to check new values and propagate their intended effects.

(3) *identification*: the assignment of identifers, unique across an entire computing community that allow multiple users to manipulate a common set of objects.

(4) *context sensitive description*: the assignment of a situation marker to values that allows alternative descriptions to coexist within a common workspace.

Our overall goal is to crossbreed Smalltalk with recent AI representation languages in order to obtain a hybrid that exhibits the strengths of both lineages. We have pursued this crossbreeding with the help and cooperation of Smalltalk's originators, the Xerox PARC Learning Research Group.

This paper first reviews Smalltalk, then discusses our implementation of each of the above capabilities within PIE, a Smalltalk system for representing and manipulating designs. We then describe our experience with PIE applied to software development and technical writing. Our conclusion is that the resulting hybrid is a viable offspring for exploring design problems.

1. Current Smalltalk

Smalltalk-76 is a programming language based on three metaphors: simulation, communication and classification. An atomic element of the language, termed an *object*, simulates a computer. It has internal state and responds to a set of instructions termed *messages*. An object

[1] Published in the Proceedings of the Lisp Conference, Stanford, California, August 1980.

responds to a message in one or all of the following ways: it changes its internal state; it transmits messages to other objects; it reads or writes an I/O channel such as the display. A sender need have no knowledge of the internal structure of a receiver: it need only know the receiver's message set. For example, there exist display objects such as rectangles that store their position and extent, and respond to messages to move, show and erase themselves.

Each object is associated with a single class. The objects associated with a given class are called its instances. The class owns a dictionary that defines methods for a set of messages. When a message is sent to an instance, that instance in turn requests the appropriate method from its class. The method returned by the class is then applied to the arguments of the message. Smalltalk has predefined classes for Rectangle and BitRect, the latter being a class that includes a state variable for storing the display state of the points enclosed by the rectangle. (Rectangle and BitRect define behavior for classes that interact with a BitMap display).

Classes are hierarchical. A superclass is used to describe the behavior common to several classes. Given superclasses, the protocol for retrieving a method is extended as follows: when a message is sent to an instance, the instance asks its class for the method associated with the message. If the class knows this method directly, it supplies it. If it does not, the class asks its superclass. If the superclass responds with a method, this method is passed back to the object. For example, BitRect is defined as a subclass of Rectangle. A method like **blink** is defined only in Rectangle since its definition, a repetitive invocation of **show** and **erase**, applies to instances of both classes. When **blink** is sent to an instance of BitRect, BitRect finds no associated method, and hence passes the buck to Rectangle, which has the desired definition.

The root of the class hierarchy tree is the class Object. If a request for a method associated with a message comes up to Object, and it does not know the definition of the message, an error occurs.

Although one class may have a great deal in common with the behavior of another, they may still differ on some methods. For example, the **show** method of BitRect differs from the **show** method of Rectangle in that BitRect displays the contents of the rectangle while Rectangle only displays the outline. The desired behavior is achieved by redefining the **show** method in the subclass. Since method retrieval is bottom up, the redefinition in BitRect will dominate the definition in Rectangle for instances of BitRect, yet be invisible to instances of Rectangle.

In addition to a method dictionary, each class also owns a list of variable names. The state of an instance is defined in terms of values for variables with these names as well as values for any variables whose names appear in the superclass chain. For example, instances of BitRect store state for *contents*, the instance variable defined in BitRect, as well as *origin* and *extent*, the instance variables defined in the superclass Rectangle. When any method of an instance is

activated by passing it a message, that activation can read and change the values of these instance variables.

A message consists of selectors and arguments. For example, the method with selector **move:** has an argument named ***distance***. A particular call to this method might look like **rect1 move: 3**, where **rect1** is an instance of class Rectangle and the argument ***distance*** is bound to 3.

The three classes, Object, BitRect, and Rectangle, appear in Figure 1 with their associated instance variables and some of their messages. The syntax employed in this and other figures of this article is for didactic purposes only, and does not correspond to Smalltalk syntax for defining classes.

The class Object with instance variables {} and methods {**is:** ***class***, ...}

> The class Rectangle, a subclass of Object, with instance variables {*origin, extent*} and methods {**show, erase, move:** ***distance*****, blink,** ...}
>
> > The class BitRect, a subclass of Rectangle, with instance variables {*contents*} and methods {**show, erase,** ...}

Figure 1. A class hierarchy in Smalltalk.

---

2. Multiple Inheritance

Smalltalk-76 does not support multiple inheritance. Classes are organized into a strict hierarchy and an instance can be associated with only one class, at a single position in the hierarchy. However, there are situations in which one desires greater descriptive power. For example, consider an environment for hardware design. Objects in this environment represent circuit elements -- resistors, chips, wires, etc. There are at least two points of view from which one may wish to examine these objects. The first is as circuit elements with associated electrical behavior; the second is as display objects that know how to draw pictures of themselves. To choose one point of view as primary, i.e., as the class of the object, and copy methods of the other points of view into this class, is clearly unsatisfactory. Equally unsatisfactory is making one class, say DisplayObject, a subclass of another, say CircuitElement. Such subclassing would be erroneous for other display objects that are not circuit elements. One would really like to be able to have multiple superclasses.

We have explored two designs for multiple inheritance. Both are based on the use of class Node, which defines the basic representational unit. An instance of Node represents some entity: a circuit part, a Smalltalk method, a paragraph of a document. Multiple inheritance is achieved by assigning perspectives to nodes. A perspective is an instance of a class that represents the node

from a particular point of view. For example, a node representing a part of a displayed circuit design might have a CircuitElement perspective and a DisplayObject perspective. Class Node defines an instance variable *perspectives* that stores each node's list of perspectives.

In our first design for multiple inheritance, the state of the object was represented entirely in the node. Perspective classes carried no state: they supplied method definitions only. This required that perspectives have backpointers to their node, since their methods manipulated the state variables stored directly in this node.

Smalltalk-76 constrains the number of named state variables to be fixed when the class is created. This is an efficiency constraint: it allows compiled code to reference instance variables by their position in a vector of fixed length rather than by their name. However, in our scheme, we prefer that it be possible to assert or delete perspectives at any time. Hence, an instance of Node cannot know all of its state variables at creation time. Our solution was to give class Node a second state variable whose value was a dictionary keyed by variable names. All variable access went through this dictionary and the dictionary could be modified at run time. Flexibility was obtained at increased computational cost. Figure 2 shows a node representing a resistor in a circuit simulation.

R17, an instance of Node, with

state* = *{ohms = 100; connection1 = wire6; connection2 = wire8; location = (100,100)}

and *perspectives* = {CircuitElement; DisplayObject}

Figure 2. A Node with multiple perspectives and a common set of state variables.

Our first design for multiple inheritance presumed that a state variable such as *ohms* had a meaning independent of the individual perspectives. Hence, it was sensible for it to be owned by the node itself. All perspectives would reference this single variable when referring to resistance. This proved adequate so long as the system designer knew all of the perspectives that might be associated with a given node, and could ensure this uniformity of intended reference.

When we extended PIE from a single user to a multiple user system, we encountered the difficulty that two users might define perspectives that employed a variable of the same name, although they had different purposes in mind for the variable. For example, one user might define a perspective InventoryPart that used the variable *location* to point to the node representing the bin containing the part, while another user might define a perspective DisplayObject that used a variable of the same name to refer to the location of the part on the screen. The result would be an unintentional clash. In our first implementation, both perspectives would be erroneously

referencing the same variable in the common pool of node variables.

Our solution was to eliminate the central database owned by the node in favor of local databases owned by each perspective. This new design achieved privacy at the cost of additional space. Furthermore, it required the user to supply functions for coordinating state variables in different perspectives that represented the same data. However, this seemed unavoidable if we were to open the process of perspective creation to multiple users. Figure 3 illustrates our representation for R17 using this second design. There is no longer a common pool of state variables.

R17, an instance of Node, with perspectives =

{A CircuitElement with *ohms* = 100, *connection1* = wire6, and *connection2* = wire8;

A DisplayObject with *location* = (100, 100);

An InventoryPart with *location* = bin101}.

Figure 3. A node with state distributed among the perspectives.

In both implementations, a message sent to a node consists of the message pattern and the class of the intended perspective. Thus, to obtain the resistance, one would execute the following statement: **(R17 as: Resistor) ohms.** The **as:** message to R17 causes R17 to return the perspective of the desired class, in this case perspective 1. Perspective 1 is then sent the message **ohms.**

An alternative to passing the perspective to the node is to require that the node poll its perspectives for any that understand the message. This approach has the advantage that the source code is more concise, but introduces the necessity to resolve cases in which more than one perspective responds to the message. This resolution could be based on a predefined ordering of the perspectives. We have not adopted this approach for two reasons: (1) In most cases, we have found that the sender knows the point of view that the recipient should employ to understand the message. (2) There is generally no good criterion for declaring that one perspective should dominate another. In those few cases where the intended perspective is not known, we have adopted the procedure that the node polls its perspectives for any that understand the message. If an ambiguity exists, a user interrupt occurs.

The use of perspectives for multiple inheritance is not new. FRL [GoldsteinRoberts77] had a scheme very much like our first implementation; KRL [BobrowWinograd77] has multiple perspectives like those of our second implementation. Both of these implementations were based on the assumption that one wants to make it easy to add a new perspective to an existing instance at any time. We have adopted this assumption in PIE.

An alternative approach is available if one allows multiple inheritance for classes, but not for instances; that is, an instance can be associated with one, and only one, class but a class can have more than one superclass. In this case, it is only in the construction of a class that clashes must be resolved between variable names occurring in more than one superclass. This is the approach employed by Thinglab [Borning77], a multiple inheritance, constraint satisfaction system.

To summarize, perspectives differ from ordinary Smalltalk objects in four respects:

- They expect to be part of a closely interacting system consisting of other perspectives and a central node; hence they come with a backpointer to their node.
- They share some of their state with other perspectives in this system, but maintain a private variable pool for their own purposes.
- They are intended to represent a point of view on an entity, rather than the entity itself.
- They can be attached at any time to a node. It is not necessary to assign all perspectives when the node is created.

3. Metadescription

Perspectives express different descriptions of the entity represented by the node. Changing these descriptions can lead to inconsistencies. We handle this problem by providing the node with various kinds of information about itself. We term this information *metadescription* to distinguish it from the primary description implicit in the node regarding the entity in the world that it represents. For a general discussion of metadescription see [BobrowWinograd77].

The first kind of metadescription we supply is knowledge of the expected type of an attribute. This information is supplied in a *constraint* dictionary. For each attribute, the constraint dictionary supplies an expression that describes the class of the expected value. For example, a value for the *ohms* attribute of the resistor perspective is expected to be of class Integer, while the value of *connection1* is expected to be a node with an associated Wire perspective. This mechanism takes care of simple unary constraints.

Secondly, we supply procedures that are triggered by the retrieval or storage of a value. These procedures typically serve to maintain consistency between dependent attributes. For example, if a change is made by the user in the connectivity of the displayed schematic, then procedures attached to the instance variables being altered can update the circuit element perspectives to correspond to the new display linkages. Similarly, attached procedures can update the inventory perspective as parts are added or deleted from the design.

To take care of less formal cases in which only the user knows what to do, we have dependency notification. A dependency list can be added to the metadescriptions of a node. The user supplies this list for a node or attribute, but does not inform the system of what actions to take if a change is made. Consequently, when the node is altered, the user is reminded of these dependencies by attached procedures, but no automatic actions are taken. For example, the user might place a dependency link between a capacitor and an inductor to serve as a reminder that the two elements are intended to operate together as a tuned circuit.

A more powerful dependency model replaces the dependency list with a pointer to a node with a *contract* perspective. The contract perspective contains a list of participants and, at a minimum, an English statement of the contract. We plan to formalize this contract progressively. For the electrical world, contracts might include the mathematical formulae that describe the circuit. For the programming domain, contracts would include the expected type of a variable. See [Borning77] for a general study of constraints as the basis of a Smalltalk system and [SussmanStallman77] for a more detailed study of dependency relations in circuits.

4. Unique Identification

The object metaphor suggests that each user of Smalltalk has his or her own unique set of objects. I run on my computer; you on yours. But the description metaphor suggests that you and I may well be working on the same set of descriptions. Hence, we need a way to separate my contributions from yours but, at the same time, to clearly identify that they are being generated to describe the same topic. To solve the first problem, we employ machinery to separate descriptions into contexts. This is discussed in the next section. To solve the second problem, we employ unique identifiers.

Consider the following scenario: I create a set of nodes representing a design and deliver these nodes to your environment for subsequent development. To accomplish this delivery, I generate a set of descriptions that can be used to recreate a set of Smalltalk objects with the same state. This was our first implementation.

However, the following difficulty arises with this scheme. You modify and supplement these nodes, and then generate a new set of descriptions. But when I reread them into my environment, how can I determine which of these descriptions should be added to existing nodes, rather than used to create a new collection of nodes?

Recognizing that two sets of descriptions describe the same intended object is a difficult problem. However, in this special case, the problem can be solved easily. A node is assigned a unique identifier when created. This identifier travels to the consumer when descriptions are

generated. The consumer checks to see if a node already exists with the identifer. If so, the descriptions of this node are appended to those already there. If no such node exists, a new node with this unique identifier is created.

The computational cost of this scheme is not excessive, since the consuming environment can maintain a table that associates identifiers with existing nodes within that environment. Hence, in consuming a set of descriptions, it is necessary only to check this table to find the preexisting node, if any. This is similar to the way Lisp atoms, or Smalltalk unique identifiers are implemented, with the important difference that the identifers are generated by the machine in such a way that two users can never create identical identifers. In fact, the identifiers consist of an encoding of the time and machine of creation.

5. Contextualization

From a design standpoint, it is important that alternative descriptions be able to coexist in the same environment at one time. Alternatives arise from a designer exploring different plans to achieve his goals; or from the interactions of several designers on a joint project. For example, one designer may propose a particular circuit to realize the specifications of a module; while another designer may propose an entirely different circuit to accomplish the same goals. In a design environment, descriptions are sensitive to who has created them and for what purpose. A user must be able to examine and manipulate such descriptions from different points of view.

To implement context sensitive descriptions, we have altered the behavior of the dictionaries that store the attribute/value pairs of perspectives. In Smalltalk-76, a dictionary is a list of attributes and an associated list of values. We have replaced the value associated with the attribute with another level of dictionary. This level of dictionary associates a *layer marker* with different values. The *layer marker* is a tag for the situation in which the value was supplied. Figure 4 shows a partial view of a layer structured description of R17.

R17, an instance of Node, with perspectives =

{A CircuitElement with *ohms* = [<layer1 100>], *connection1* = [<layer1 wire6>], and *connection2* = [<layer1 wire8> <layer2 wire13];

A DisplayObject with *location* = [<layer1 (100,100)> <layer2 (300, 300)]}

Figure 4 A partial view of the node *R17* with layers indicated. Layer1 stores the original design. Layer2 stores a change in the display location of the resistor and an associated change in the circuit connectivity.

Storage and retrieval is therefore situation dependent. Storage is done with respect to a layer. Retrieval is done with respect to a sequence of layers. The retrieval algorithm checks the layers in order for a value, returning the first value in the layer sequence. This layer sequence is called a *context.* These notions of layer and context are derived from Conniver [Sussman72]. There are minor differences in the implementation, and major differences in the use of the mechanism. This is discussed in more detail in [BobrowGoldstein80].

Values stored in a layer represent a coordinated set of values. Suppose the connectivity of R17 in a circuit is changed as a display object. An attached procedure (or the user) might make the corresponding change in the circuit simulation. These two changes are meant to be coordinated, and are therefore placed in the same layer. By "coordinated", we mean that one sees either both changes or neither in any view of the circuit. All retrievals in a context will get either both these values (if the layer is included in the context) or neither.

The flexibility to represent alternative descriptions in layers comes at the cost of increased complexity. We have designed several display interfaces to explore different mechanisms for simplifying the presentation of this inherently more complex database. For example, one interface provides a way for a user to view two different contexts simultaneously with differences between the two highlighted. We have also explored the use of metadescription to default some of the contextual choices that would otherwise fall on the user, e.g., selecting the default layer for assertions and the default context for retrieval. Finally, we have supplied commands that suppress the context machinery. The user stores and retrieves state in a context free fashion. This is faster, occupies less space, and has no cognitive overhead for remembering alternative contexts. But the user no longer can explore alternatives or separate his contributions from those of a codesigner. All three of these strategies have proved useful in some circumstances, but it remains an important research goal to make the context machinery available to the user in a convenient fashion.

6. Use of PIE

The PIE system provides an environment for doing software development. Perspectives are provided for representing Smalltalk classes and methods. A user of PIE is therefore able to build a collection of nodes that represent a software system. Unique identifers and contexts allow users to engage in cooperative design and to explore alternatives. When a design is complete, it can be installed in Smalltalk by generating executable code from the node descriptions. Other designs described in separate contexts remain unaffected by this installation. Metadescription is used to express type knowledge regarding method variables, thereby obtaining the strengths of a typed language while still preserving the underlying flexibility of an untyped interpreter.

The utility of this descriptive base for developing software is illustrated by the following

experiments: (1) We have successfully redesigned PIE's user interface within PIE. Ordinarily, such redesigns would clobber the coding environment itself, but the separation between description and installed code prevents such conflict. (2) We are able to describe a method as belonging to multiple classes, despite the fact that the Smalltalk kernel does not allow this. At the descriptive level, a node representing a method may be linked to more than one class. Within Smalltalk itself, a method is local to a class. For compatibility, all that is necessary is that installation of the description involves placing copies of the compiled code in each class. However, at the descriptive level, the designer can treat the method as a single integral entity; editing it affects its occurrence in all of its classes. (3) Multiple perspectives and metadescription support improved browsing and prettyprinting of code, thereby improving the user's ability to examine his designs. (4) Unique identifiers and contexts provide a mechanism for generating an incremental system release. The new system is created by transmitting a layer with the changes to a consumer and then asking the consumer to do a reinstallation. Separating release changes into layers allows the consumer to examine the alterations of the release and exercise some choice regarding which parts he wishes to accept, before performing the reinstallation.

The same machinery has also been used to support a document design environment. Nodes are used to represent the structure of the document; i.e., the document is a tree of nodes whose root represents the document as a whole and whose terminals are the individual paragraphs. The nonterminals of the tree are chapters, sections and sub-sections. Again, contexts and identifers facilitate coauthoring and exploring alternative organizations, two capabilities not well supported by present text editing environments. Metadescription can be used to express formatting constraints. Multiple perspectives allow a paper to appear as either an abstract, a citation, a bibliographic reference, the outline for a lecture, or a formatted document, depending on the desired point of view.

The PIE system code occupies approximately 200 kilobytes and 100 pages of listing in a Smalltalk system of approximately 1 megabyte and 1000 pages of listing. Storage space for nodes grows as layers increase, and previous or alternative values for attributes of nodes are stored. Retrieval time increases with the number of layers in the retrieval context. However, neither price has proved exorbitant since PIE has been used largely as an interactive design tool. In this application, time is primarily limited by the responses of the user, i.e. there is more thinking than computing. Space is released when the design is complete and an installed package of code is created.

7. Conclusion

We conclude by reconsidering Smalltalk's underlying metaphors of simulation, communication and classification in the light of our addition of descriptive machinery to the language.

In Smalltalk-76, objects simulate computers and therefore have a fixed identity. They use a predetermined set of state variables and respond to a fixed set of messages. In PIE, nodes have a flexible set of state variables which can grow or shrink as the attributes of individual perspectives are changed. Furthermore, the message set can change as new perspectives are supplied or old perspectives deleted. Nodes are more analogous to an evolving biological species than to an inanimate computer. At any moment in time, a member of the species has a fixed anatomy and physiology. Over time, however, both the anatomy and physiology evolve.

In Smalltalk-76, objects have an unambiguous message semantics. A message is sent to an object and that object, in turn, requests the appropriate method from its class. In PIE, nodes have multiple perspectives and more than one perspective may supply a method for a given message. The user must specify the perspective, or allow the node to decide. Communication is still an applicable metaphor, but the complexity of communication has increased as the underlying objects have moved from a monolithic to a pluralistic society.

In Smalltalk-76, objects participate in a simple, hierarchical classification scheme. In PIE, nodes are the locus of a set of descriptions and behaviors, each generated from a different point of view. Classification, with its implication of simple hierarchy, has been replaced by description, with its more open-ended connotation.

Thus, the evolution from Smalltalk to PIE has produced a change in the behavior of the basic computing element. In Smalltalk, objects have a fixed structure and engage in communication based on a simple classification scheme. In PIE, nodes have an evolving structure and engage in a more complex communication based on the use of descriptions. We believe that this evolution yields a more flexible environment for exploring design problems.

References

1. Bobrow, Daniel G. and Goldstein, I.P. "Representing Design Alternatives". *Proceedings of the AISB Conference*, Amsterdam, 1980

2. Bobrow, Daniel G., Terry Winograd, and the KRL Research Group. "Experience with KRL-0: One Cycle of a Knowledge Representation Language". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 1977, pp. 213-222.

3. Borning, A. "ThingLab -- an Object-Oriented System for Building Simulations Using

Constraints". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, 1977, pp. 497-458

4. Dahl, O.J. and Nygaard, K. "SIMULA--an ALGOL-Based Simulation Language", *CACM* 9, September 1966, pp. 671-678.

5. Goldstein, I.P. and Roberts, R.B. "NUDGE, A Knowledge-Based Scheduling Program". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, 1977, pp. 257-263.

6. Ingalls, Daniel H. "The Smalltalk-76 Programming System: Design and Implementation". *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 9-16.

7. Kay, A. "A Personal Computer for Children of All Ages". *Proceedings of the ACM National Conference*, August 1972.

8. Sussman, G. and McDermott, D. "From PLANNER to CONNIVER - A Genetic Approach". *Fall Joint Computer Conference.* Montvale, New Jersey, 1972.

9. Sussman, G. and Stallman, R. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis". *Artificial Intelligence, 9*, 1977, pp. 135-196.

REPRESENTING DESIGN ALTERNATIVES[1]

1. Introduction

A major activity in artificial intelligence research is the design of complex systems. Yet most software environments do not support this activity well. They do not allow within the system description of different properties of a design nor the flexible examination of alternative designs. All designers create alternative solutions, develop them to various degrees, compare their properties, then choose among them. Yet most software environments do not allow alternative definitions of procedures and data structures to exist simultaneously; nor do they provide a representation for the evolution of a particular set of definitions across time. It is our hypothesis that a context-structured database can substantially improve the programmer's ability to manage the evolution of his software designs.

Present computing environments support the creation of alternative designs only with file services. Typically users record significant alternatives in files of different names; the evolution of a given alternative is recorded in files of the same name with different version numbers. We contend that this use of files provides both an impoverished structure as well as an inflexible one. The poverty is a result of the fact that file names are simply a limited length sequence of characters, hardly an adequate scheme to describe the purpose and contents of a file, and its relation to other files. It can be an adequate reminder to the originator of the name, but is often opaque to a new reader. The rigidity is a reflection of the fact that one typically cannot use parts of files as part of a new composite design, except by tedious text editing. Finally, the most serious limitation is that files are "off-line" in the sense that the alternative designs are not stored within the computing environment in a form that can be easily manipulated by the programmer. Although Interlisp [Teitelman, 78] provides some facilities for manipulating pieces of a file (e.g. individual function definitions), it still suffers from the "off-line" limitation.

To ameliorate this software bottleneck, we have constructed a computing environment in which "on-line" descriptions of alternative software designs can be readily created and manipulated. We use a context-structured description-centered database to describe code. Such databases have been explored in artificial intelligence research for over a decade as a mechanism to represent alternative world views. [e.g. Hewitt, 71; Sussman & McDermott, 72].

1 Published in the Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior, Amsterdam, July 1980.

Our application of this machinery is novel in several respects. (1) Previous applications have focussed on the use of such databases by mechanical problem solvers. We are exploring the use of such databases in a mixed-initiative fashion with the user primarily responsible for their creation and maintenance. (2) Previous applications have always demanded a uniform overhead in space and time for adopting the context machinery. We are exploring configurations for a design environment that allow the programmer to trade flexibility for efficiency, decreasing the system's investment in tracking the evolution of particular parts of a design at the price of not being able to represent alternatives simultaneously in primary memory. Thus, employing the design environment is not an all or nothing choice for the user. (3) Previous applications have been to problems of limited complexity. In our application of context structured databases to software design, we are exploring their utility in a world several orders of magnitude more complex.

To understand the pros and cons of context structured environments for software design, we have implemented a prototype environment and conducted several experiments. The environment is called PIE, an acronym for personal information environment. PIE allows the user to build context sensitive descriptions of code, documents, and, indeed, any object for which a machine representation exists. PIE has been employed (1) to allow a programmer to create alternative software designs, examine their properties, then choose one as the production version, (2) to coordinate the interactive design of two programmers, and (3) to coordinate the documentation and definitions of an evolving package of code.

2. The Smalltalk Environment

To describe PIE further, we must first introduce Smalltalk [Ingalls, 78; Kay, 74], the programming environment in which it has been implemented. Smalltalk is an object-oriented programming language. (See Dahl & Nygaard [66] on Simula and Hewitt et al [73] on "actors" for related work on such programming languages). Behavior arises from the transmission of messages between objects. Each object is, in essence, a simulation of a computer. It can respond to some number of messages and it maintains its own state between message invocations.

The message set of an object is specified by Smalltalk's class structure. Each object is an instance of a class. When a message is sent to the object, it asks its class for the method associated with that message. The class either contains the definition directly, or if not, passes the request to its superclass. For the object to understand the message, its definition must occur somewhere in this superclass chain. Thus, objects of the same class are analogous to computer products of the same model.

Figure 1 shows a fragment of the definition of a Smalltalk class for Spaceship. The fragment shown indicates that instances of Spaceship understand messages that simulate motion and collision and that each instance carries its own private state regarding its position and velocity.

```
Class new title: Spaceship
    superClass: Object "class Object is the root of the superClass hierarchy."
    declare: 'allSpaceships' "a class variable --shared by all instances"
    fields: 'position velocity' "instance variables -- each instance has private versions of these"

Moving "methods are divided into 'protocols' -- this one is called Moving"

accelerate: dv "dv is the argument of the method with selector accelerate"
    [velocity←velocity+dv]

move [position←position+velocity. "points understand the message +"
        self crashes => "self refers to this instance. => indicates a conditional expression"
            [↑ self explode] "if condition is true, move returns with value of self explode"
        self display. "done if condition is false -- display is a message this instance understands"]

Collisions "another protocol"

  crashes | ship "ship is a local variable for the activiation"
"This assumes that all ships are of unit size, and collide only when at the same point"
[for: ship from: allSpaceships do: [ ship collideAt: position =>[↑true]].↑false]

  collideAt: place
    "a method to test if I collide with another object at place."
[position=place =>[↑true] ↑false]
```

Figure 1: Partial Definition of a Smalltalk class

We chose Smalltalk over Lisp, the usual vehicle for AI research, because Smalltalk has a superior set of interactive display facilities. DLISP [Teitelman, 77] provides enough capabilities we believe, but was not available on the same fast hardware. These interactive display facilities were of critical importance to allow the functionality of the design environment to be delivered to a user. No matter how powerful the design tools, no experiments would have been possible with an interface based on an inadequate communication channel. Using Smalltalk, however, has required that we reimplement machinery common to such AI languages as FRL [Goldstein & Roberts, 77] and KRL [Bobrow et al, 77]. This has proved straightforward because the object oriented structure of Smalltalk is congenial to the frame-based viewpoint of a AI representation languages.

3. The PIE Environment

To describe Smalltalk code, we created a class of Smalltalk objects called *nodes*. Nodes are analogous to KRL units, or FRL frames: they consist of a set of attribute value pairs with support for attached procedures, the use of defaults, meta-descriptions and inheritance.

PIE provides convenient ways of viewing relationships between nodes, and viewing and changing the properties of nodes. One can automatically create nodes which describe existing pieces of the Smalltalk system, and conversely, make the system congruent with a description of it. Node23 in Figure 2 is a description that might have been been computed from one method of the Smalltalk code shown in Figure 1.

```
Node23
    class           Node17      "Node17 is the node describing the class Spaceship"
    selector        'crashes    "This is a unique string -- like a Lisp Atom"
    localVariables  ('ship)     "This is a set of unique strings"
    variablesUsed   ('ship 'allSpaceships 'position 'mySize)
    methodBody      "This is an editable paragraph"
        [for: ship from: allSpaceships
          do: [ ship collideAt: position =>[↑true]].↑false]
    comment
        'This assumes that all ships are of unit size, and collide
        only when at the same point'
```

Figure 2. A node describing the method for crashes

In PIE, changing the values of any of these attributes does not automatically change the object being described by the node. The node describes an intended object in the system, not necessarily the version that exists in the system. This is worth emphasizing as one of the principles characterizing our point of view towards the design process.

> ★ **The Description Principle:** In a system there should exist a descriptive level at which objects can be described without actually affecting the objects themselves.

4. Representing Alternative Designs

Using node structure, there are two distinct ways to have alternative descriptions of the same object: coreference and context. We have explored both, with our current preference being for the use of contexts.

Coreference uses separate nodes to describe separate alternatives. In Figure 3, **Node25** is a description of an alternative version of **crashes**. The intended identity of the **Node23** and **Node25** (they are both are describing the same object) is made explicit with the coreferentNodes

attribute.

```
Node25
   class          Node18          "Node18 is the node describing the class Spaceship which differs
                                   from Node17 in having an additional instance variable -- mySize"
   selector 'crashes
   localVariables   ('ship)
   variablesUsed    ('ship 'allSpaceships 'position 'mySize)
   methodBody       "a different method body"
      [for: ship from: allSpaceships
         do: [ ship collideAt: position of: mySize =>[↑true]].↑false]
   comment    'Uses mySize for each ship to determine overlap'
   coreferentNodes (Node23)
```

Figure 3. An alternative method for crashes

However, coreference has certain difficulties. The first is that it does not represent the manner in which two descriptions may differ on some attributes but otherwise be identical. The second is that the coordination of the choice of Node23 vs. Node25 and other choices in the system for consistency is not expressed. For this reason we have chosen to explore another way of expressing alternatives.

In this second method, all descriptions (values of attributes) of any node are relative to a context. *Context* as we use the term extends the notion of context as used in Conniver [Sussman & McDermott, 72], and has certain similarities to the vistas of partitioned semantic nets [Hendrix, 75].

★ **The Context Principle:** All attribute-values in the system are relative to a context, and alternatives in a system are expressed by alternative contexts.

When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible.

5. Incremental Design

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. Every programmer is aware that software has a life cycle: following its birth, it undergoes progressive refinement in response to changing external requirements. PIE supports the incremental modification of a design by providing a fine structure to contexts that we have not, as yet, discussed.

A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer.

Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Figure 4 shows a layer C containing some coordinated changes to the spaceship class of Figure 1. This layer contains those changes necessary to allow the class to use size information in determining collisions. In a context which contained this layer dominating those containing the information implicit in Figure 1, the changes would be visible. Those attribute-values such as the superclass of Spaceship that are not contained in layer C would be found in less dominant layers.

```
Node17 "the node for the class Spaceship"
    fields:     ('position 'velocity 'mySize)  "a change in a declaration"
    methods     (... Node23 Node27 ...)

Node23 "the node for the method crashes"
    methodBody
        [for: ship from: allSpaceships
         do: [ ship collideAt: position of: mySize =>[↑true]].↑false]

Node27 "the node for the method that tests for a collision"
    selector    'collideAt:of:
    methodBody
        [(position+mySize>place-size)and:(position-mySize<place+size) =>[↑true]
         ↑false]
```

Figure 4. Layer C, containing coordinated changes to use mySize

---

Figure 5 shows several spaceship nodes in which the values of attributes have not been filtered by a context sensitive lookup. Instead, we see the underlying data structure, which is an association list of layers and values. Layer B is the base layer in which all the nodes were presumed to have been originally defined for this example.

```
Node17 "the node for the class Spaceship"
    fields:  LayerB ('position 'velocity)
             LayerC ('position 'velocity 'mySize)
Node23 "the node for the method crashes"
    methodBody
     LayerB
        [for: ship from: allSpaceships
         do: [ ship collideAt: position  =>[↑true]].↑false]

    LayerC
        [for: ship from: allSpaceships
         do: [ ship collideAt: position of: mySize =>[↑true]].↑false]
```

Figure 5. An unlayered view of node structure

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context each time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Given the existence of layers, a complex design developed over many stages can be summarized into a single new layer. The old layers, reflecting past choices, can then be deleted. Thus, the designer, if he wishes, can compress the past, achieving a more compact representation at the price of no longer representing the dynamics of the design.

6. Coordinating Designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment with these properties: (1) *non-interference.* Two designs may overlap. It must be possible to examine the overlap without the designs overwriting one another. (2) *incompleteness.* It must not be necessary for a design to be complete before it is examined. (3) *merging.* It must be convenient to create a common design from the individual contributions. It was encouraging for us to learn that the context/layer machinery created to manage alternatives lent itself well to meeting these requirements for coordinating partial designs.

Non-interference between the overlap of two partial designs was accomplished by adopting the convention that different designers place their contributions in separate layers. Thus, where an overlap occurred, the divergent values for some common attributes were separated by distinct layers. Handling incomplete designs of software was facilitated by the distinction between intensional node descriptions and the actual code definitions. Since the node descriptions were not installed code, they could be partial and hence non-executable with no difficulty.

Merging two designs can be viewed as a process that creates a new layer into which are placed the desired values for attributes as selected from two or more competing contexts. It is hence very much like the summarization process described earlier, but it is relative to more than one context and requires user interaction. For complex designs, the merge process is, of course, non-trivial. We do not, and indeed cannot, claim that PIE eliminates this complexity. What it does provides is a more finely grained descriptive structure than files in which to manipulate the pieces of the design.

Understanding how to merge two designs is facilitated by examining commentary supplied by the designers regarding the rationale of their choices. But this raises the classic software problem of coordinating documentation with design. Fortunately no additional machinery is required in PIE to address this problem. Commentary such as the rationale of a procedure, or its dependencies on other procedures, can be stored as attribute value pairs within the node describing the procedure in question. A request to be informed of the rationale of some change is answered by fetching this information from the same layer as the one which records the change, thus keeping them coordinated. Figure 4 shows how the rationales of various method definitions are recorded in the layer along with the altered definitions.

7. Complexity.

We claimed in the introduction that PIE copes with problems several orders of magnitude more complex than those previously represented in AI systems such as Conniver. By complexity we mean both the size of the data base in the system, and the variety of operations done on contexts. The Conniver database was never efficient enough to implement any usable subsystems. McDermott's [McDermott, 74] examination of the Monkey and Bananas problem within Conniver exercised it to its limit.

PIE is able to build a context sensitive description of any class within Smalltalk. Thus, it can be applied to any programming problem that a Smalltalk programmer undertakes. This is analogous to using Conniver to build a programmer's interface to Lisp. Attacking problems of this size is, in part, possible because we have more computational resources than were available in the early 70's. PIE runs as a stand alone job on a processor with at least the

power of a KA10. However, it is also possible because we have implemented machinery to allow the programmer to move between context sensitive and context free descriptions at will. Thus, there is a more congenial marriage between PIE and Smalltalk than there was between Lisp and Conniver. This is discussed in the next section.

An interesting side effect of PIE's ability to describe any code within Smalltalk is that it can and has been used to describe itself. Thus, PIE's present capabilities have passed the test of being sufficiently powerful to support its own development, for example, by allowing us to examine alternative implementations of the PIE user interface within PIE.

8. Efficiency versus Flexibility

PIE allows the user to trade flexibility for efficiency. At one extreme, the user can employ standard Smalltalk mechanisms for defining new code. If this route is chosen, then no evolutionary history is maintained, and no context overhead is paid. However, if the user wishes to pay the price of some decrease in efficiency of storage and retrieval time, then he can first build a set of nodes describing Smalltalk code, then continue his development in a context structured fashion. From this point forward, the evolutionary history is maintained. If the user reaches the point where he once again prefers efficiency to flexibility, the context definitions can be converted to pure Smalltalk and the layers deleted. If desired, the user can first store the layers remotely, preserving the ability to recreate the context description later. All these facilities are curently implemented.

This discussion suggests how a central design facility can serve as the nucleus of a network of remote servers that provide current packages to users. Periodically, the design server can release new layers to these servers with updates to particular designs. The servers can then generate new Smalltalk versions and release these designs to clients. Clients who wish to know what has changed, can get a description from the new layer.

9. Interaction

PIE's ability to represent non-trivial alternative designs raises deep problems related to the user interface. How can we make available this power in a useable form? What are the cognitive requirements of the programmer? Presently we are employing an interface modelled on the standard Smalltalk interface for examining and altering code. This interface, called the browser, displays a hierarchy of descriptions of Smalltalk code to the user. The user can examine any method by a process of selection that specifies first a category of classes, then a particular class, then a protocol of methods within the class, and finally a particular method. This scheme of organizing code into a four-level taxonomy has been adopted in PIE to minimize the overhead for a Smalltalk user learning to employ the PIE environment. However,

PIE makes this classification context dependent. As with the standard Smalltalk browser, the user can alter the definitions of any object viewed. But these alterations are made in the dominant layer of the associated context, and do not affect the Smalltalk kernel itself, whereas making changes with the standard Smalltalk browser forces immediate incorporation of any changes.

Research is needed to explore whether this interface is adequate given the increased complexity of a context structured environment. In Smalltalk, the hierarchy of code definitions is the primary structural organization. In PIE, this hierarchy is now context dependent. Has this additional complexity made the Smalltalk organization inadequate? Will we need a classification scheme with more levels of division, or will some other kind of organization be appropriate? Just one of the problems that we will have to consider is that in a design environment, there is no need for a particular method description to be associated with only a single class, even though the actual Smalltalk system requires that the method be separately compiled for each class to which it belongs. Hence, a strict hierarchy is obviously inadequate.

9. Conclusions

This paper presents only a sketch of the PIE system; our research is reported in greater detail in Goldstein & Bobrow [80]. We have not discussed here issues in the design of the user interface, although a successful interface is critical to delivery of these capabilities to the user. We only suggest here that layered networks are applicable to more than software: an extended example in cooperative writing of a document is given in the larger work. Finally, the system has as yet had only limited use. We do not know which features will be used most, which need to be automated to be helpful, and which may prove to be too complex to be useful. Recording and analyzing this experience is an important part of our research program.

A major theme of Artificial Intelligence research has been the development of languages to describe complex evolving structures. In general, these structures have been the belief structures of an artificial being about some subject matter (e.g., the SRI consultant's [Hart, 75] beliefs about the state of a water pump being constructed, or SAM's [Schank et al, 75] beliefs about what went on in a story it just read). We have been exploring the premise that these techniques can be used to describe the complex evolving structure of a software system, and as such can provide aids to the designer of such a system. One use of artificial intelligence is to amplify human intelligence. We suggest that the (recursive) application of AI techniques to AI can have a powerful effect on the development of the field.

References

1. Bobrow, Daniel G., Winograd, Terry, and the KRL Research Group, Experience with KRL-0: One cycle of a Knowledge Representation Language, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977, 213-222.

2. Dahl, O.J., and Nygaard, K., SIMULA--an ALGOL-Based Simulation Language, *CACM* 9, September 1966, 671-678.

3. Goldstein, I.P. and Bobrow, D.G., A Layered Approach to Software Design, *Xerox PARC* CSL-5-80, 1980.

4. Goldstein, I.P. and Roberts, R.B., NUDGE, A Knowledge-Based Scheduling Program, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977, 257-263.

5. Hart, P., Progress on a Computer Based Consultant, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi: 1975, 831-841.

6. Hendrix, Gary G., Expanding the utility of semantic networks through partitioning, *Advance papers of the fourth international joint conference on artificial intelligence*, Tbilisi: 1975, 115-121.

7. Hewitt, C., Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot, Ph.D. Thesis, June 1971 (Reprinted in AI-TR-258 MIT-AI Laboratory, April 1972.)

8. Hewitt, C., Bishop, P., and Steiger, R., A universal modular ACTOR formalism for artificial intelligence, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, 235-245.

9. Ingalls, Daniel H., The Smalltalk-76 Programming System: Design and Implementation, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ: January 1978, 9-16.

10. Kay, A., *SMALLTALK, A communication medium for children of all ages*, Palo Alto, CA: Xerox Palo Alto Research Center, Systems Science Laboratory, 1974.

11. McDermott, D.V., *Assimilation of new information by a natural language-understanding system*, AI-TR-291 MIT-AI Laboratory, February 1974.

12. Schank, Roger, and the Yale AI Project, *SAM—A story understander*, Yale University, Computer Science Research Report #43, August 1975.

13. Sussman, G., and McDermott, D., From PLANNER to CONNIVER—A genetic approach, *Fall Joint Computer Conference*, Montvale, NJ: AFIPS Press, 1972.

14. Teitelman, W., *Interlisp reference manual*, Palo Alto, CA: Xerox Palo Alto Research Center, Computer Science Laboratory, 1978.

15. Teitelman, W., A display oriented programmer's assistant, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977.

BROWSING IN A PROGRAMMING ENVIRONMENT[1]

1. Introduction

A browser is a software development tool that supports the incremental examination of a system by accessing some kind of information network. A user starts at a canonical place in this network, and selects entities that represent parts of the system. This causes the browser to display the substructure of the system connected to the selected entity, and some information about that entity. In this manner, a browser can be employed to engage in a hierarchical examination of a system by proceeding level by level from subsystem to module to sub-module, until the terminal structure—possibly individual procedure definitions—is reached. In addition, the browser allows a user to add or alter structure at any point in this examination process.

Most programming environments allow a user to retrieve and manipulate different parts of a software system, if the programmer knows their exact name and location; but do not support well the examination of structure whose exact description the programmer does not know. In such situations, the programmer will frequently be reduced to examining file directories, hoping that the file names reveal the contents of the file. A browser seeks to ameliorate this difficulty by allowing a user to examine different regions of a software system based on their general classification. Thus, the underlying database imposes an organization on the software system analogous to the organization imposed on a library by the Dewey decimal system. The browser provides an electronic analog of moving from a general classification to the stacks, and then subsequently browsing there.

Browsers were introduced into Smalltalk by Larry Tesler in 1977, and have since become a mainstay of the Smalltalk programming environment. (The general nature and goals of Smalltalk are described in Kay [77]; the 1976 implementation in Ingalls [78]; and the Smalltalk browser in Goldberg and Robson [79].) In recent research, we have extended the simple, hierarchical system model provided by Smalltalk and developed a generalization of the Smalltalk browser to manipulate these richer descriptions [GoldsteinBobrow80a,b,c; BobrowGoldstein80]. We have dubbed this extended environment PIE, an acronym for Personal Information Environment.

In the next two sections, we describe the Smalltalk system model and its associated browser. This is followed by two sections that describe the PIE system model and its browser. The following nine questions are used as a framework for comparing the functionality of these two browsers.

[1] Published in the Proceedings of the 14th Hawaii Conference on System Science, January 1981.

1) *Overview:* How much of the information network can the user see at one time?

2) *Path:* What part of his path to the current position is visible to the user?

3) *Presentation:* What should be displayed on the screen for each selection?

4) *Operations:* What operations can be performed on the view for each selection?

5) *Multiple Views:* Can more than one view of the network be seen? Are they all of the same form?

6) *Consistency:* What guarantees of consistency are there between multiple views?

7) *Alternative Access*: Can the user find a known entity in the system without tracking through the network?

8) *Integration*: Is the data environment integrated with the operational environment of the underlying system?

9) *Changeability*: Can the user change the format in which information is displayed?

2. The Smalltalk System Model

Smalltalk is an object oriented programming system, where behavior arises from the transmission of messages between objects. Objects are grouped into *classes*, all of which have identical internal structure, and respond to the same set of messages. An object is like a simulation of a computer; it can respond to set of instructions, maintaining its state between invocations. Smalltalk generalizes Simula67 [Birtwistle73] and is related to the Actor languages developed by C. Hewitt [Hewitt73].

The Smalltalk information network partitions all classes into *categories* for ease of access. These categories are not mutually exclusive, although multiple category membership is generally avoided. (Since classes are stored in files corresponding to their category, multiple category membership gives rise to redundant storage and possible inconsistencies between versions.) A *method* is the code which implements the class specific response to a message. The set of methods of each class is partitioned into mutually exclusive groups called *protocols*. Neither categories nor protocols has any significance for the Smalltalk interpreter; rather they are artifacts of the desire to browse through the system.

There is a subclass hierarchy in the Smalltalk system that does have semantic significance. A class can inherit behavior and structural description from another class called its superclass. All instances of a particular class contain the fields specified in the superclass. If the subclass has no specialized behavior (method) for responding to a particular message, it will request that its superclass respond to the message. This inheritance is a very powerful way of sharing behavior.

3. The Smalltalk Browser

Figure 1 shows a sequence of views of a Smalltalk browser as a user selects a path through the network. The browser is a rectangular region on the display screen called a *window* and is built from 6 sub-windows called *panes*. The top pane is the title pane and shows the label 'Smalltalk Browser'. Below it is a row of four *list panes* that display, from left to right, categories, classes, protocols and methods. The lower pane is a *text pane* that displays text associated with the most recently selected item.

Figure 1a shows the browser in its initial state with the leftmost list pane displaying part of the list of categories defining the Smalltalk system. The pane can be *scrolled* to view other categories in the list. The browser enters the state shown in Figure 1b in response to the user selecting the category Data Structures. A selection is made by moving a cursor over the item to be selected and depressing a button on the device controlling the cursor. Selections appear in inverted video in the actual system, but are shown in boldface in the figures. The most recent selection is in bold italics. The selection of Data Structures causes the classes of this category to be displayed in the second list pane and a template for defining a new class to appear in the text pane. In Figure 1c, the user selects Set, a class whose instances provide the behavior of sets by appropriately manipulating an array. This selection causes the class' protocols to be displayed in the third list pane and the definition of the class to appear in the text pane. The user can edit this definition to modify the title, superclass, or fields of the class. In Figure 1d, the user selects the Access protocol, causing its methods to appear in the last list pane and a template for defining new methods to appear below. In Figure 1e, the user selects the has: element method and its definition appears in the text pane. Figure 2 shows the path that the user has traversed in the system taxonomy. (This particular graphic view is not generated by Smalltalk.)

The organization entries under categories and protocols are not actually items of that type, but rather data structures that can be edited to alter the taxonomy. For this reason, the organization entries are not shown in Figure 2. Changing the category organization by selecting it and editing the text that appears below can move existing classes to different categories. The protocol organization serves a similar function for its class.

3.1 Overview

The browser shows a slice of the four level system taxonomy that extends through all four levels but is of limited breadth. Figure 2 shows this slice relative to a graphic view of the taxonomy. At his discretion, the user can select any element in the displayed slice of the taxonomy. To see other elements on a given level, the user must *scroll* that pane, thereby changing the slice of the tree seen in the pane.

3.2 Path

Since the hierarchy is only four deep, the user can see the entire path from the root. The user cannot see, and the browser does not maintain, a history of other nodes that have been selected before, but are not on the path.

3.3 Presentation

Selection causes text and sub-structure to be displayed. Sub-structure is displayed in the list pane to the right. Text consisting of either templates or definitions is displayed below. For categories and protocols, a template is shown for defining new classes and methods respectively; for classes and methods, their definition appears. The reason for this difference is that categories and protocols have no semantic significance other than grouping a set of subordinate elements.

3.4 Operations

For each of the list panes, operations are defined for deleting, printing and filing the selected element. These commands are available from a menu that is not shown.

Insertion is not an explicit menu command. Instead, it occurs in two different ways. New classes and methods are inserted in their respective categories or protocols as a side effect of compiling their definitions. Old classes and methods can be rearranged by manipulating the table that the browser presents when the **organization** entry is selected in the category or protocol pane. Manipulating this table is also the mechanism for creating new categories and protocols.

A limitation is that the browser does not permit the creation of partially defined classes or methods. A class or method must be compilable to be successfully included in a category or protocol; this is a result of the browser assumption that the data structure it is viewing is the one currently installed in the system. This has undesirable consequences for program design when the designer wishes to delay certain decisions. In this respect, the marriage between the browser and the software environment is too intimate.

3.5 Multiple Views

Several browsers can be brought to the screen at once and can overlap. Commands are provided to move a browser to a new region of the screen and to view an obscured browser. The result is that the display screen is like a desktop with multiple browsers representing different pieces of paper.

This browser provides a command to spawn additional text windows that display the selected method. These windows maintain a constant view of the method, allowing the user to browse to other regions of the network. They are incomplete views of the method, however, in that they do not display its class or protocol, and hence these attributes of the method cannot be altered through this window.

The hardcopy format of Smalltalk code represents a third view of the system. This view is a depth first listing of the tree. Users occasionally prefer this view to the browser in order to obtain a perspective on a segment of code. The hardcopy format cannot be manipulated within the system.

The browser does not support other taxonomic views of the system such as an examination of the class/subclass hierarchy.

3.6 Consistency

The view seen on one browser is almost completely independent of that seen on a second, *even if they are both looking at the same method or class definition.** This means that if a method is changed using one browser, the definition seen on the screen for the other is not altered because that browser is unaware that the underlying model it is viewing has changed since it fetched the definition. Only if an explicit request is made to fetch the definition again is the underlying model queried, thereby ensuring that the view is consistent.

* The exception is that browsers do check whether the list of classes has changed whenever they are reactivated. If a class has been added or deleted from this list, the browsers reenters its initial state. No check is made for changes to the definitions of existing classes, protocols, or methods.

The reason for the inconsistency is two-fold. First, the view in the browser is just that, a computed view, and changes to that view are not reflected immediately in the model. Only when the method is compiled is the underlying system model altered. This is desirable since the user should be able to complete a set of changes to a procedure before it is altered permanently. Otherwise compilation might be attempted on an inconsistent state. Second, when a change does occur to some software object, there is no way for that object to inform the appropriate views since the underlying system model has no knowledge of existing views.

There are at least two solutions to this problem. One is to give each object responsibility for updating views of itself, using a "notification protocol"; for example, a class whose method changes would notify all browsers which have informed it of their current interest. A second solution is to give each view the responsibility for keeping itself updated, and to provide a way for it to check what the last time an object it is viewing changed. Then any time a viewer becomes active, it can compare its last update time with this list to see if updating is required.

3.7 Alternative Access

The only means to move through the network is by progressive selection of displayed objects. No browser commands exist to select an object via a partial description or even by specifying its name.

3.8 Integration

The browser does not support access to other kinds of data such as manuals, primers, and system specifications nor does it support examination and manipulation of instances of classes.

The browser is integrated in a limited fashion with a history list of changes in the sense that defining or redefining methods affects this list. However, deleting a method has no effect on the history nor can the history list be examined through the browser. No distinction is made between different kinds of modifications such as the difference between adding a breakpoint and making a permanent change made to the code.

3.9 Changeability

The user can change the size, number and position of browsers on the display screen by invoking commands supplied by the browser, but no commands are supplied to alter the relative widths of various panes.

The user can alter the behavior of the browser in two ways. He can redefine methods in the browser (using the browser itself), although bugs in these changes could make the interface inoperative. Or he can subclass the classes used to define the browser and make whatever changes he wishes in these subclasses. This is a safer strategy, since old style browsers are unaffected, but all behavioral changes must be programmed in Smalltalk itself. It is equally parsimonious in that subclasses inherit all of the behavior of their superclasses, except for messages that they define directly.

The browser does not support idiosyncratic behavior for particular objects of a given type: all classes, for example, are treated identically.

4. Summary of Smalltalk Browser Strengths and Weaknesses

4.1 Strengths

The Smalltalk browser provides an excellent way of examining and editing the Smalltalk system code as evidenced by its universal adoption within the Smalltalk community and relative stability. Its browsing capabilities and the associated system architecture of a taxonomy of constructs serve a useful documentation role. Users often familiarize themselves with new software by browsing through new categories in a system release. The browser provides a uniform way to examine and manipulate the software, and guides novices with templates for creating new entities.

4.2 Weaknesses

The Smalltalk browser keeps no history of its interactions except for the names of methods that have been changed. It only reflects the current state of the world; there is no way to go back and forth between different consistent states. The system does not help a user to maintain any design constraints other than the ones implicit in the programming language. For example, a programmer cannot indicate that two methods in a class are dependent, and that subsequent modifications to one should be checked for compatibility with the other. There is no incremental

way of modifying the behavior of the browser by attaching your own procedure to provide a specialized function in the interface; for example, one cannot provide specialized templates for new methods of a particular class.

The Smalltalk browser also reflects deficiencies in the underlying system model. Smalltalk provides for comments for classes and methods but not for categories of classes or protocols of methods. Class comments are separately manipulable from the class definition; method comments are not. Storing a method comment requires that the procedure be recompiled.

5. The PIE System Model

PIE was motivated, in part, by the goal of providing a more complete and more integrated representation for Smalltalk systems. It provides a network structured database whose nodes describe all the entities in the system and employs techniques developed for describing entities in knowledge representation languages like KRL [BobrowWinograd77].

Nodes provide a uniform way of describing entities of many sizes, from a small piece such as a single procedure to a much larger conceptual entity. For example, nodes are used to describe code in individual methods, classes, categories of classes, and configurations of the system to do a particular job. Sharing structures between configurations is made natural and efficient by sharing regions of the network.

The uniform use of node structure extends to software documentation. Manuals and specifications can be embedded in the network using nodes representing the chapters, sections and paragraphs of the material and can be cross-linked to the relevant software. Because software and documentation coexist in the same environment, it is easier to develop them in a coordinated manner.

Nodes are distinct from the system objects that they represent. Changing a node does not immediately alter its corresponding software object. For example, the node representing a class can be created and a partial definition supplied. This node can be stored, examined and edited. It does not affect the underlying Smalltalk environment, however, until its description is compiled.

Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. For example, the **structuralSpec** of a Smalltalk class defines the structure of each instance by specifying the fields it must contain; the **proceduralSpec** defines the protocols; the **interfaceSpec** defines the set of messages required by external clients, and the **documentSpec** describes the implementation and its use.

Perspectives may provide partial views which are not necessarily independent. For example, the **proceduralSpec** and the **interfaceSpec** both describe certain methods of the class. Attached procedures are used to maintain consistency between such perspectives.

Each perspective supplies a set of specialized actions appropriate to its point of view. For example, the *print* action of the structuralSpec perspective of a class knows how to prettyprint its fields and class variables, whereas the proceduralSpec perspective knows how to prettyprint the methods of the class. These actions are implemented directly through messages understood by the Smalltalk classes defining the perspective.

All values of attributes of a perspective are relative to a *context*. Context as we use the term derives from Conniver [SussmanMcDermott72]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is possible to create alternative contexts in which different values are stored for attributes of various nodes. For nodes representing software, these contexts typically describe alternative designs. One can compare and test alternatives without leaving the design environment.

Contexts are themselves nodes in the network. This allows a description of the rationale for the set of changes to be stored in the context node in the network, in the same way that descriptions for for a method node contain comments on their purpose.

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ *contracts* between nodes to describe these dependencies. These contracts are themselves nodes with specialized behaviors. These behaviors include installation of procedures to maintain consistency of simple constraints expressed in a formal language, and notification to the user when changes have been made to contract participants. Use of contracts raises a number of questions which we have just begun to explore; e.g. when should one check agreements and still avoid seeing temporary states of inconsistency during the process of change.

Finally, the PIE system provides perspectives which allow the system to describe itself. Perspectives themselves are described in the system, and small modifications to the behavior of a particular perspective can be made by manipulation of the network structure. Nodes can be assigned meta-nodes whose purpose is to describe defaults, constraints, and other information about their object node. Information in the meta-node is used to resolve ambiguities when a message is sent to a node having multiple perspectives.

6. The PIE Browser

The PIE browser was constructed as a generalization of the Smalltalk browser, in order to minimize the overhead of Smalltalk users immigrating into the PIE environment. It is shown in Figure 3a. Two additional panes have been added in the middle of the browser. The left pane lists the perspectives of the most recently selected node while the right pane lists the attributes of the selected perspective. The title pane shows the node at which the browsing begins and the context from which the network is being viewed.

In Figure 3b, the user has selected the node representing the Data Structures category. This causes the two perspectives of this node to be displayed. The first is the perspective describing categories: it includes a classes attribute and additional attributes describing the most recent file and modification dates to classes in the category. The second is the **description** perspective, common to many nodes, that specifies a title and optional text for the node. In this case, the text attribute is employed to store a comment regarding the category, and this comment is displayed in the text pane.

In Figure 3c, the user has selected the **category** perspective and its attributes appear in the attribute pane. In Figure 3d, the classes attribute is selected and its value, a list of nodes representing the classes of this category, appears in the second list pane. The attribute is used as a label for the pane. In Figure 3e, the user has selected the Set node, and its perspectives appear below. Thus, moving from one node to the next in the network requires selection of a node, then a perspective, then an attribute. Figure 4 shows a graphic representation of the PIE network and the path traversed by the user.

6.1 Overview

As with the Smalltalk browser, the user can see a slice of the network. In addition to nodes surrounding previous selections, this slice includes the perspectives and attributes of the current selection. We have explored browsers that show the perspectives and attributes of every node in the path, but these trade breadth of view for increasing complexity on the screen.

The labels on the four upper list panes are dynamic and computed from the selection. The Smalltalk browser employed static labels since the same attribute was always displayed in a given list pane.

6.2 Path

The PIE network is not restricted to a depth of four. However, the PIE browser contains only four list panes, a constraint derived from the size of the screen. To go deeper into the network, the user can shift the view to the left. In Figure 5, the user has moved the view one to the left. The origin of the browser is now the Data Structures category and the rightmost pane is available to show subordinate nodes linked to the **has: element** method. In this case, the user is examining nodes representing constraints on the definition of the method. If the user tried to see substructure which would logically be to the right of the fourth pane, PIE blinks the browser to indicate that it cannot show the requested information in the current browser configuration. The user can then shift the view as described, or spawn a new browser rooted further down the tree, and continue.

The PIE browser does not maintain a chronological history of selections. Hence, it is limited, like the Smalltalk browser, to displaying only four steps in the path to the current selection. An unfortunate consequence of this lack of historical information is that while the view can be shifted to any node in the network, the browser cannot recreate selections made from that node. Hence, a

shift to the right, for example, from the **Data Structures** node back to the **Code** node, would require that the user remake his selection choices to again be examining the **has: element** method.

6.3 Presentation

To minimize the interactions required by the user, the browser can operate in a mode in which it makes various default decisions on its own initiative. These decisions are based on additional descriptions provided in the network. For example, the network contains descriptions that specify that the **category** perspective should be selected by default over the **description** perspective and that its **classes** attribute should be displayed. As a result of these default specifications, the selections of Figures 3c and 3d are made by the system and selecting the data structures category in Figure 3b produces the display of Figure 3e immediately. Hence, the user need not engage in any more interaction with the PIE browser than with the Smalltalk browser to conduct similar actions. The user can override these defaults by making explicit perspective or attribute selections.

The specification of the default display behavior of a node is described in *meta-nodes* linked to perspective types and to particular nodes. In the former case, the meta-node applies to all instances of the perspective. In the latter case, its advice is idiosyncratic to a particular node. These meta-nodes can be examined and edited from the browser.

Templates for creating new nodes of a particular type are available upon request and are stored in the meta-node of the perspective. They are shown automatically only if they are specified to be the default display information. Many perspectives, not just those for classes and methods, have templates.

6.4 Operations

The PIE browser supplies four standard operations: insertion, deletion, filing and printing. Insertion consists of adding a node to the list and assigning it a perspective. Default knowledge is employed to supply a particular perspective when the list is constrained to be a set of nodes of a particular kind. For example, the **classes** attribute of the **category** perspective has the default description that all of its elements have a **class** perspective assigned. Descriptions of nodes can be stored without having to compile them. Therefore partial descriptions of methods can be left in the network and returned to later.

Insertion of nodes of arbitrary type eliminates the need for an *organization* entry. Categories and protocols are created by adding nodes with those perspectives. Rearranging an old organization is accomplished by moving nodes from one attribute set to another.

The PIE browser also differs from the Smalltalk browser in that special actions specific to perspectives at a node can be invoked by the user through a special menu. This menu is computed from the selected node, using default description that specifies a subset of the messages of a

perspective to be user commands. The PIE browser can view nodes with arbitrary perspectives in any pane. Hence, the ability to interrogate the perspective for its associated commands was necessary. Since the Smalltalk browser views only four kinds of objects and these objects are tied to particular panes, this generality was not included.

6.5 Multiple views

There are three different senses in which multiple views are available to the user of PIE. The first is similar to that of the Smalltalk browser. There can be more than one instance of a browser on the screen at a time, viewing different parts of the Smalltalk system.

A second kind of multiple view comes from the notions of context embodied in the PIE network. The value of any attribute is context dependent. The user can change the view seen in the browser by changing the context associated with that particular browser. This causes the browser to recompute all fields seen.

The third arises from the fact that the user can request an outline view to be generated of the substructure of the selected node. A portion of the subtree descending from the selected node is shown in an indented outline format. The default perspective and attribute of each node is used to determine which part of the subtree to display. For class Set, this outline would include the Set node, the protocol nodes of its structuralSpec perspective, and the method nodes of each protocol. This outline is very close to the standard hardcopy view of Smalltalk code—a fact that is not accidental. The defaults have been chosen to make this view the preferred one.

6.6 Consistency

As with the Smalltalk browser, there are no backpointers from nodes to views. This means that a change made to the network through one browser is not reflected in another browser's view computed earlier. One approach to solving this problem is presently being introduced into Smalltalk by providing backpointers from software objects to their views. A separate control process is assigned responsibility for maintaining consistency. Another approach that we are considering is to describe the browser itself in the PIE network in order to take advantage of the contract machinery provided by PIE to maintain consistency between descriptions. However, this is still an unexplored area.

6.7 Alternative Access

A browser provides one way to get access to a node in an information network. Sometimes it is useful to shift the point of view of the system to a node which matches a given description without having to browse through one level at a time. This is provided in PIE. A user can specify the perspective type and some distinguishing features of a node. For example, he can search for classes entitled Set, any class that is a subclass of these classes, or even any class whose comment includes the substring 'set'. PIE engages in a a search and causes the view to be shifted to the

selected node. If more than one node matches the description, PIE offers the user all matches. Selection of a match causes the view to be shifted to the selected node.

Some indexing facilities are provided to limit the potential candidates for a match: each perspective maintains a list of the nodes to which it has been assigned. This is a very simple scheme, but the present size of the Smalltalk system—consisting of several hundred classes owning several thousand methods—does not require anything more elaborate.

One novelty of our searching machinery with respect to traditional database design is that no general set of indices are maintained. Rather, each perspective has its own matching protocol. Thus, if a perspective receives a description like 'set' without a specification of the attribute of the perspective to which this description must match, the perspective itself decides which attributes can be used as the basis of a match. For example, the structuralSpec perspective checks the title and superclass attributes, but not the field variable or class variable declarations. This is in contrast to most data base environments where entities are matched against a pattern by a standard algorithm which matches the values of attributes, perhaps using range tests. Because PIE is integrated in the Smalltalk system, each entity can run its own idiosyncratic program to test whether it matches a description.

6.8 Integration

The PIE browser integrates the examination of data, code, documentation, and system description since all of this information is uniformly described in the network. The browser also integrates the computation of views of the database with the underlying programming language. In most data bases, "views" are supported which compute virtual relations from real ones that exist in the data base. However, the programming language to compute these views is impoverished, usually being restricted to expressions in the relational calculus. The advantage of this language is that it makes the update problem easier by providing an expression calculus with no side effects for specifying how to compute a view each time. In PIE, the full power of the Smalltalk language is available, but we must provide notification and time stamp mechanisms to help with the update problems.

6.9 Changeability

In addition to the ways that the Smalltalk browser can be altered, the behavior of the PIE browser is affected by changes to the information network. A user can alter the default display behavior of perspectives by editing the meta-nodes involved. For example, the user can change the meta-node to cause the default text displayed when a class is selected to be the comment describing the class rather than the class definition.

7. Summary of PIE Browser Strengths and Costs

7.1 Strengths

Some strengths of the PIE browser arise from the improvements in the PIE system model over the standard Smalltalk model. The network database that the browser manipulates is arbitrarily deep, allows multiple perspectives and context-sensitive description, integrates the representation of text and software, and supports search and matching behavior. Other strengths arise from the availability in the network of interface-specific description. This includes description of default perspectives and attributes for display, and idiosyncratic behavior of particular entities. This self-description minimizes the user's workload for expected actions.

7.2 Weaknesses

The PIE browser shares a number of weaknesses with the Smalltalk browser. For example, it does not maintain a history of user interactions and it does not provide any means to maintain consistency between multiple views. However, the PIE model provides a possible solution to both of these weaknesses. Nodes can be employed to represent the history of a design and to represent contracts between multiple views. This solution has the appeal of building upon existing machinery and maintaining a highly integrated system model. These are current research issues for us.

Another potential weakness common to both the Smalltalk and the PIE browsers is that they do not present the network in a two dimensional graphical notation such as the one shown in Figures 2 and 4. Indeed, since those figures were used to elucidate the network structure being examined by the browsers, one might very well ask why it is not the format actually generated by the interfaces. The answer, of course, is that the pane-oriented structure of both browsers is simpler to implement than a general two-dimensional layout program. However, a research issue is whether this implementation simplicity comes at a serious cost in comprehensibility to the user. Experiments need to be performed with users of different levels of expertise to investigate which graphical metaphors are most useful in clarifying the presentation of a network description of software.

8. Conclusions

PIE reflects a natural evolution of the Smalltalk system model to provide a more extensive description of an evolving software design. The PIE browser has evolved in parallel. An unexpected result is that the boundaries between the two have become fuzzy as the network describing the software system is employed to describe the desired display behavior. Specifications of system semantics do not usually include such descriptions. However, the availability of more powerful machines, coupled to the increasing complexity of software, makes their inclusion both possible and necessary.

The PIE system and its associated browser is largely independent of the semantic details of Smalltalk. It is based on the existence of a network description of a software system. It could be the basis for programming environments for other software languages, to the extent that those languages supported display facilities and a network database which can hold representations of code easily accessible by the language processors. Experiments reported in [Cattell80] are planned for exploring these ideas in a programming environment for Mesa, a PASCAL-derived systems programming language.

References

1. Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, C., *Simula Begin*, Auerbach, Philadelphia, 1973.
2. Bobrow, D.G. and Goldstein, I.P. "Representing Design Alternatives", *Proceedings of the AISB Conference*, Amsterdam, 1980.
3. Bobrow, D.G. and Winograd, T. "An overview of KRL, a knowledge representation language", *Cognitive Science* 1, 1 1977.
4. Cattell, R.G.G., "Integrating a Database System and Programming/Information Environment", to appear in a *Joint Issue of SIGMOD, SIGPLAN, and SIGART*, October, 1980.
5. Goldberg, A. and Robson, D. "A Metaphor for User Interface Design", *Proceedings of the 13th Hawaii International Conference on System Science*, Jan. 1979, pp. 148-157.
6. Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference*. Stanford University, 1980a.
7. Goldstein, I.P. and Bobrow, D.G., "A Layered Approach to Software Design", *Xerox PARC CSL-5-80*, 1980b.
8. Goldstein, I.P. and Bobrow, D.G., "Descriptions for a Programming Environment", *Proceedings of the First Annual Conference of the American Association for Artificial Intelligence, August*, 1980c.
9. Hewitt, C., Bishop, P., and Steiger, R., "A Universal Modular ACTOR formalism for artificial intelligence", *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, pp. 235-245.
10. Ingalls, Daniel H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 9-16.
11. Kay, A. "Microelectronics and the Personal Computer" *Scientific American*, September, 1977.
12. Kay, A. and Goldberg, A. "Personal Dynamic Media" *IEEE Computer*, March, 1977.
13. Sussman, G., & McDermott, D. "From PLANNER to CONNIVER -- A genetic approach", *Fall Joint Computer Conference*, Montvale, N. J., AFIPS Press, 1972.
14. Thacker, C. P. McCreight, E. M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. "Alto: A personal computer" in Siewiorek, Bell and Newell, *Computer Structures: Readings and Examples*, 1980.

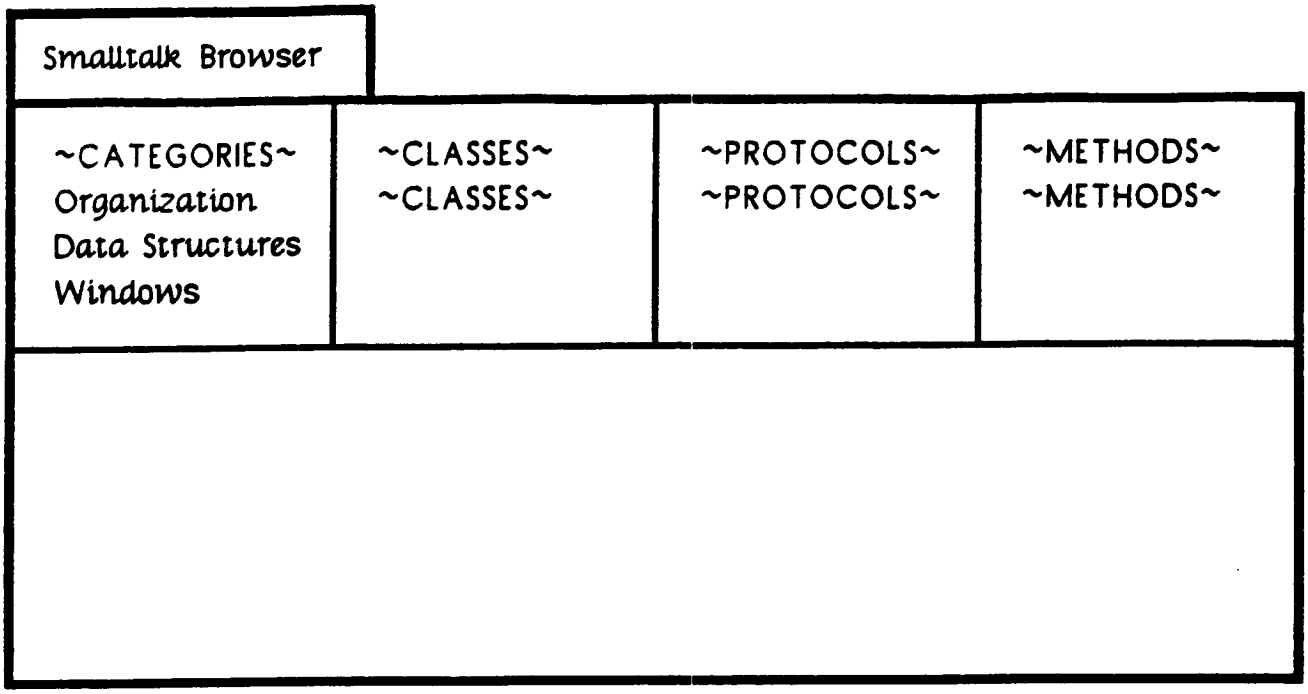

Fig. 1a. The browser is in its initial state, displaying a list of categories.

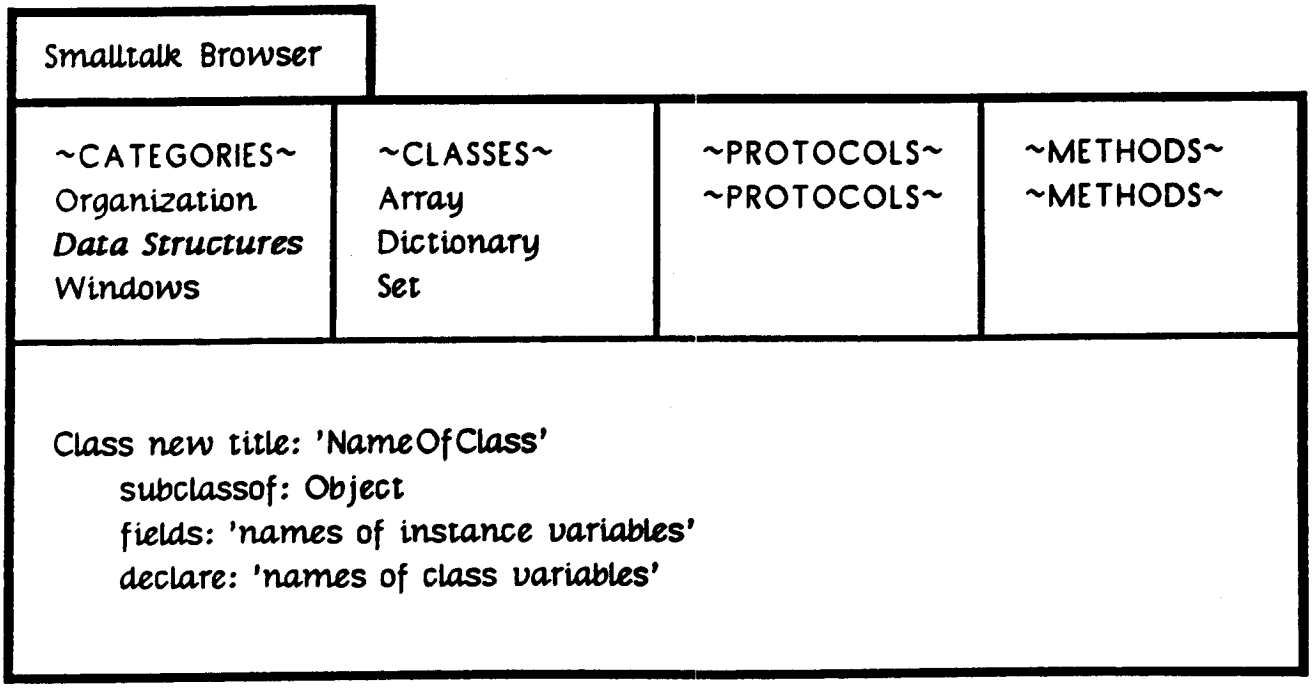

Fig. 1b. The user has selected the Data Structures category. The classes of this category appear in the classes pane and a template for defining new classes appears in the text pane.

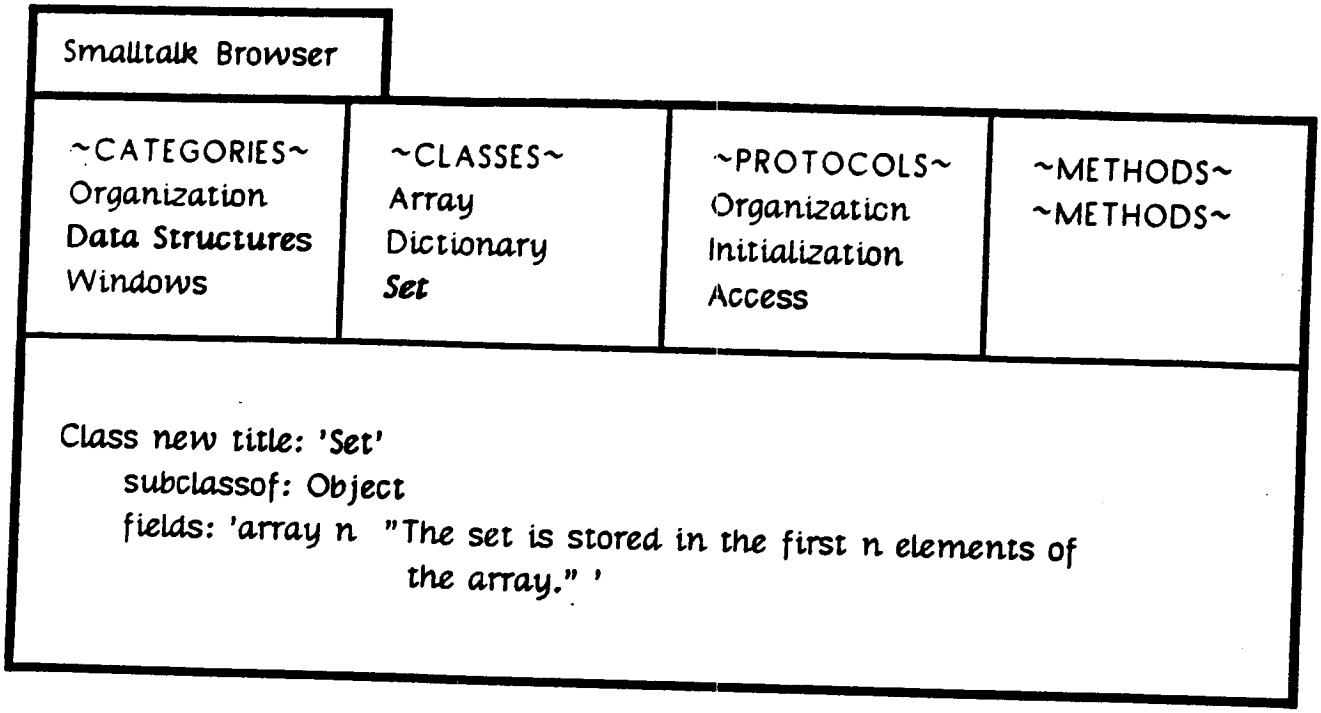

Fig. 1c. The user has selected the class Set.
The protocols of this class appear in the Protocols pane
and the definition of the class appears in the text pane.

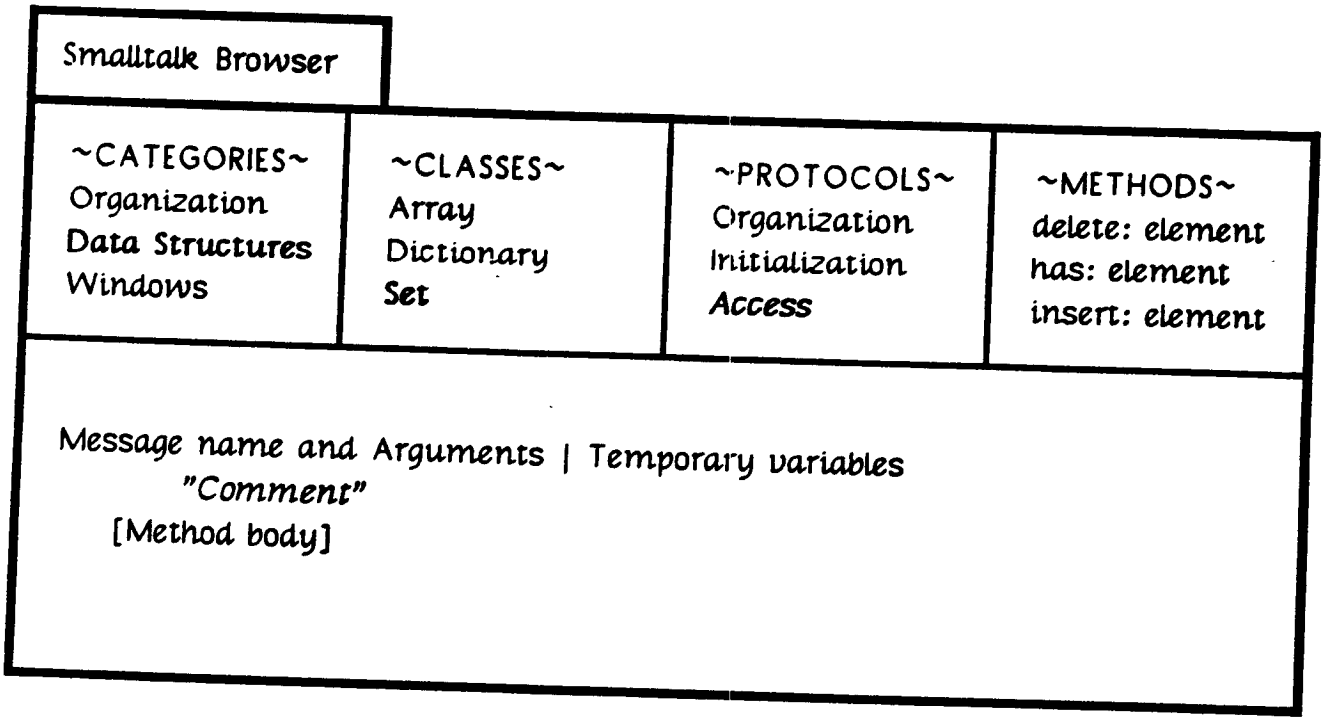

Fig. 1d. The user has selected the Access protocol.
The methods of this protocol appear in the Methods pane
and a template for defining new methods appears in the text pane.

Messages understood by perspectives represent one of the advantages obtained from developing a knowledge representation language within an object-oriented environment. In most knowledge representation languages, procedures can be attached to attributes. Messages constitute a generalization: they are attached to the perspective as a whole. Furthermore, the machinery of the object language allows these messages to be defined locally for the perspective. Lisp would insist on global functions names.

4. Contexts and Layers

All values of attributes of a perspective are relative to a *context.* Context as we use the term derives from Conniver [SussmanMcDermott72]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is natural to create alternative contexts in which different values are stored for attributes in a number of nodes. The user can then examine these alternative designs, or compare them without leaving the design environment. Since there is an explicit model of the differences between contexts, PIE can highlight differences between designs. PIE also provides tools for the user to choose or create appropriate values for merging two designs.

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer. Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context the first time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Layers and contexts are themselves nodes in the network. Describing layers in the network allows the user to build a description of the rationale for the set of coordinated changes stored in the layer in the same fashion as he builds descriptions for any other node in the network. Contexts provide a way of grouping the incremental changes, and describing the rationale for the group as a whole. Describing contexts in the network also allows the layers of a context to themselves be asserted in a context sensitive fashion (since all descriptions in the network are context-sensitive). As a result, super-contexts can be created that act as *big switches* for altering designs by altering the layers of many sub-contexts.

5. Contracts and Constraints

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ contracts between nodes to describe these dependencies. Implementing contracts raises issues involving 1) the knowledge of which elements are dependent; 2) the way of specifying the agreement; 3) the method of enforcement of the agreement; 4) the time when the agreement is to be enforced.

PIE provides a number of different mechanisms for expressing and implementing contracts. At the implementation level, the user can attach a procedure to any attribute of a perspective, (see BobrowWinograd77 for a fuller discussion of attached procedures); this allows change of one attribute to update corresponding values of others. At a higher level, one can write simple constraints in the description language (e.g. two attributes should always have identical values), specifying the dependent attributes. The system creates attached procedures that maintain the constraint.

There are constraints and contracts which cannot now be expressed in any formal language. Hence, we want to be able to express that a set of participants are interdependent, but not be required to give a formal predicate specifying the contract. PIE allows us to do this. Attached procedures are created for such contracts that notify the user if any of the participants change, but which do not take any action on their own to maintain consistency. Text can be attached to such informal contracts that is displayed to the user when the contract is triggered. This provides a useful inter-programmer means of communication and preserves a *failsoft* quality of the environment when formal descriptions are not available.

Ordinarily such non-formal contracts would be of little interest in artificial intelligence. They are, after all, outside the comprehension of a reasoning program. However, our thrust has been to build towards an artificially intelligent system through successive stages of man-machine symbiosis. This approach has the advantage that it allows us to observe human reasoning in the controlled setting of interacting with the system. Furthermore, it allows us to investigate a direction generally not taken in AI applications: namely the design of memory-support rather than reasoning-support systems.

An issue in contract maintenance is deciding when to allow a contract to interrupt the user or to propagate consistency modifications. We use the closure of a layer as the time when contracts are checked. The notion is that a layer is intended to contain a set of consistent values. While the user is working within a layer, the system is generally in an inconsistent state. Closing a layer is an operation that declares that the layer is complete. After contracts are checked, a closed layer is immutable. Subsequent changes must be made in new layers appended to the appropraiate contexts.

6. Coordinating designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment in which potentially

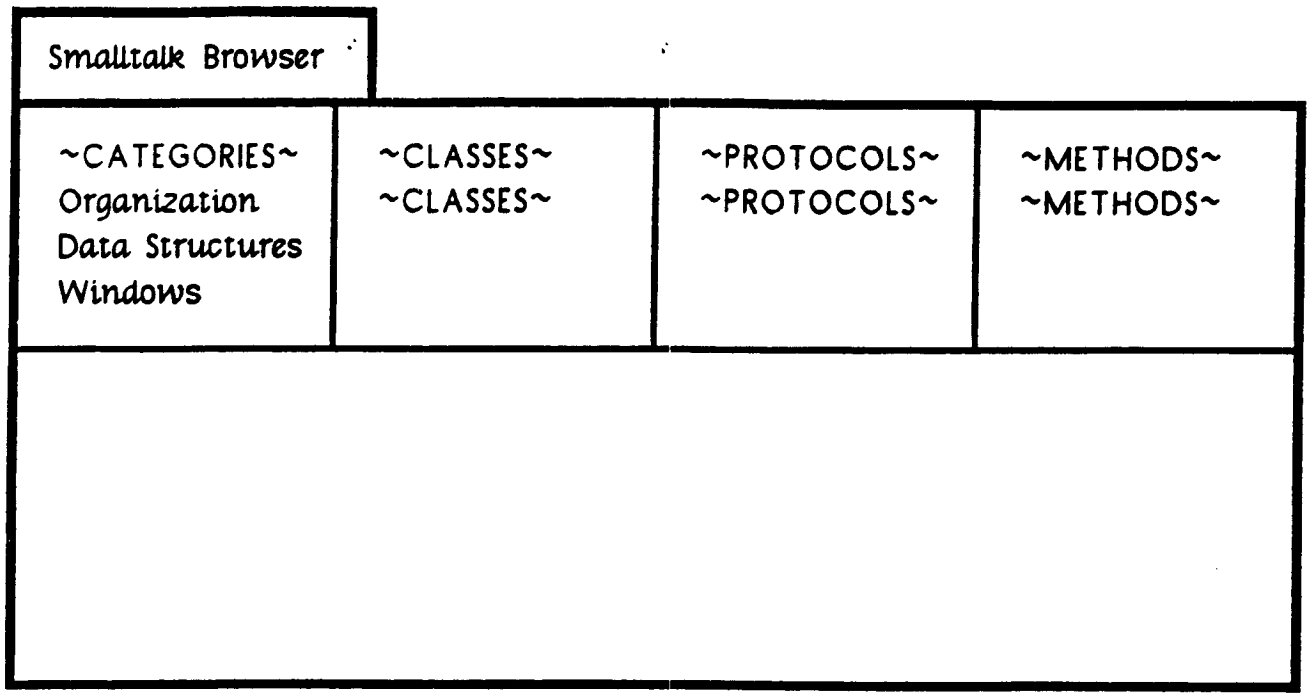

Fig. 1a. The browser is in its initial state, displaying a list of categories.

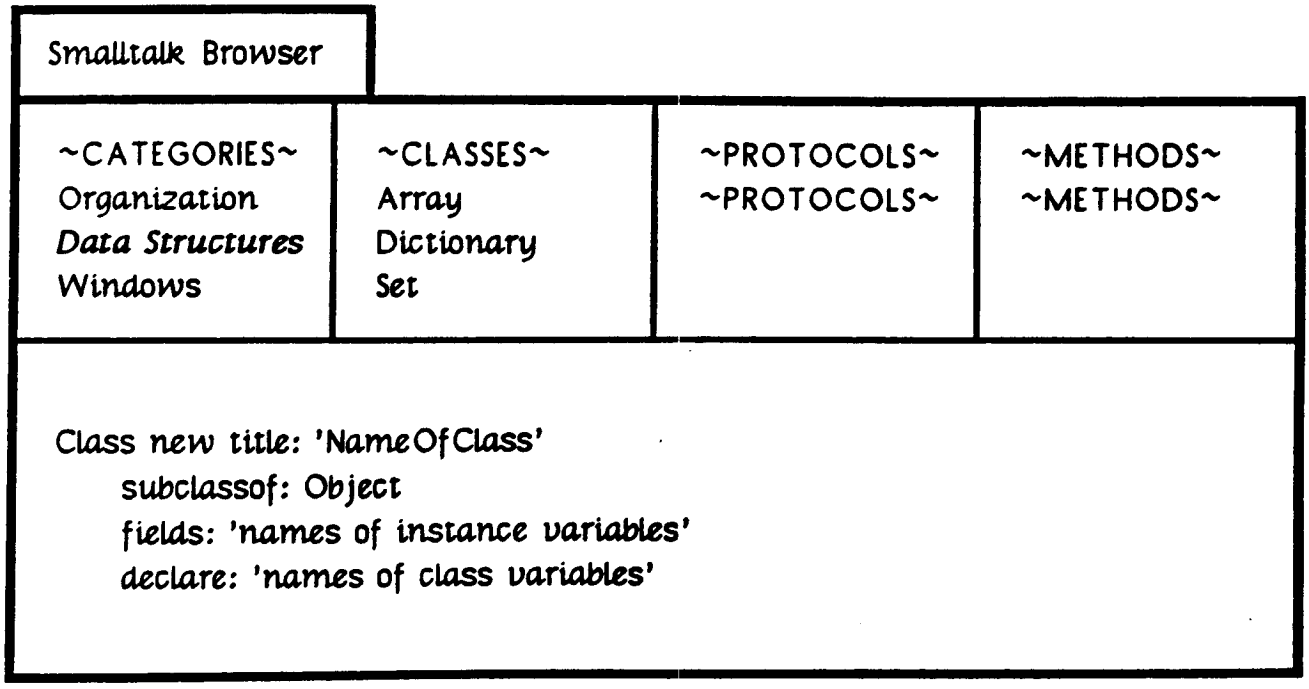

Fig. 1b. The user has selected the Data Structures category. The classes of this category appear in the classes pane and a template for defining new classes appears in the text pane.

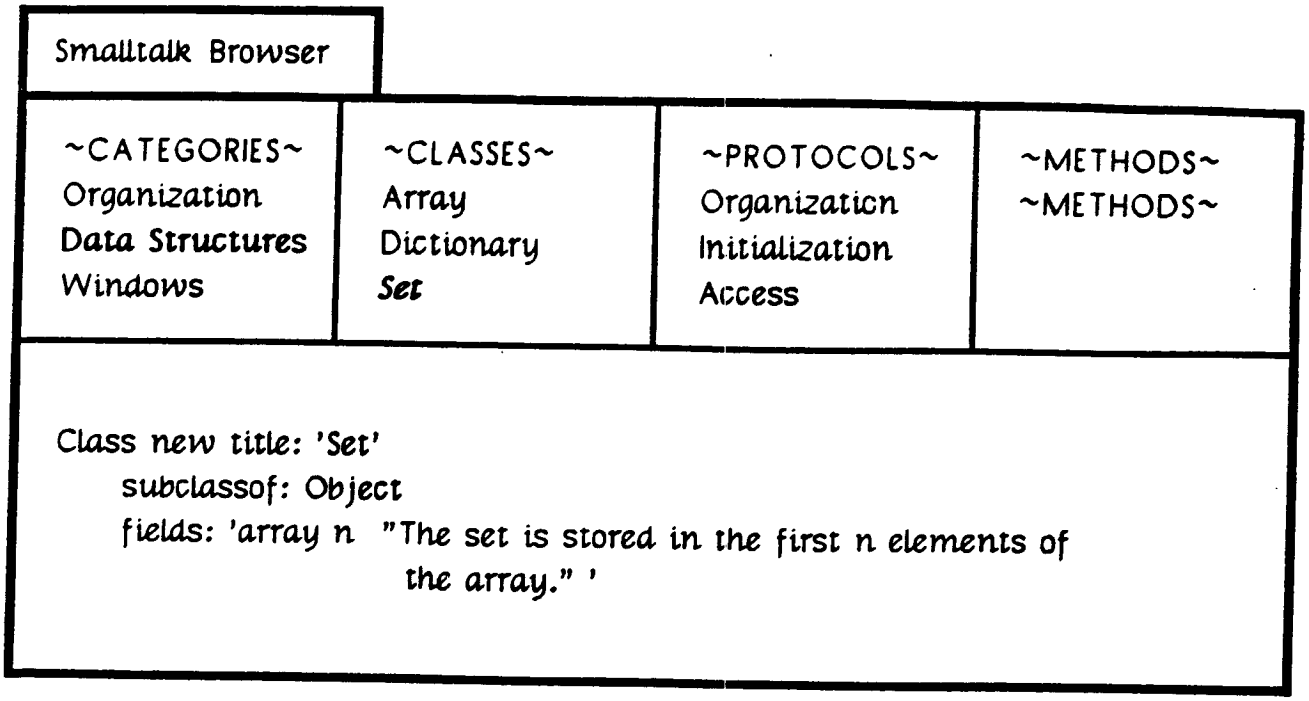

Fig. 1c. The user has selected the class Set.
The protocols of this class appear in the Protocols pane
and the definition of the class appears in the text pane.

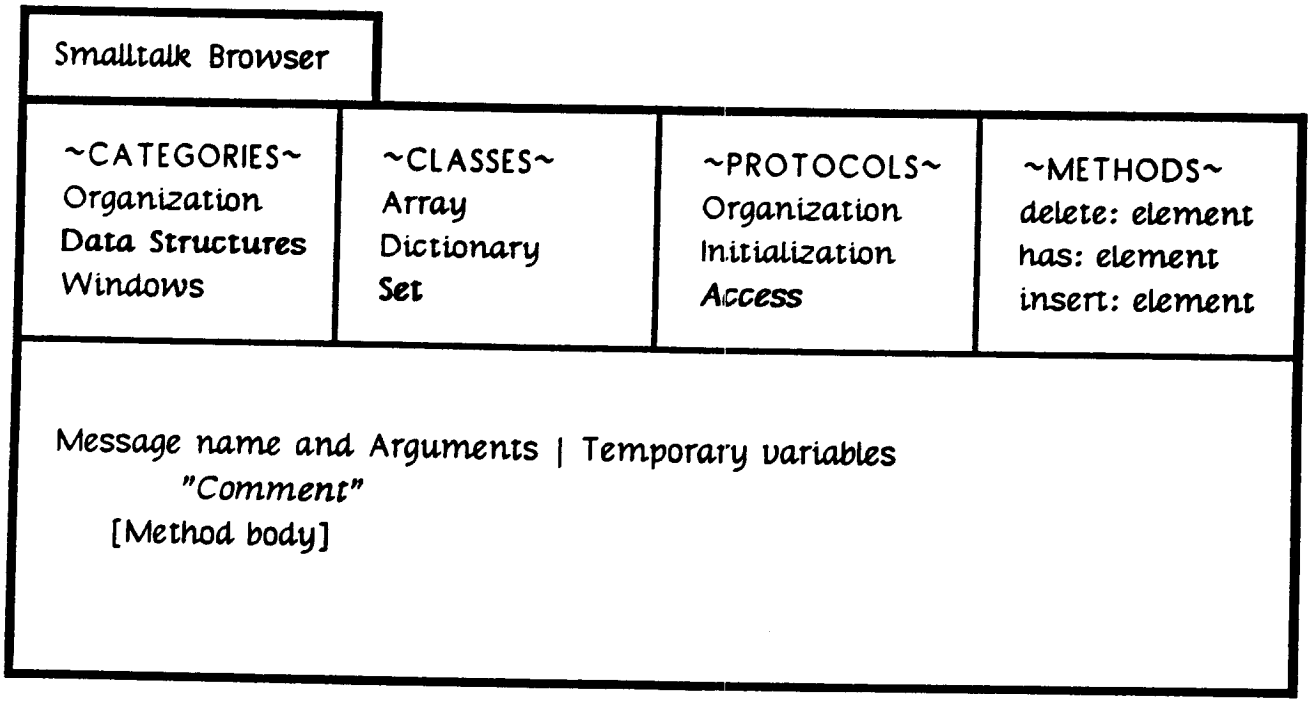

Fig. 1d. The user has selected the Access protocol.
The methods of this protocol appear in the Methods pane
and a template for defining new methods appears in the text pane.

Smalltalk Browser

~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Organization	Array	Organization	*delete: element*
Data Structures	Dictionary	Initialization	***has: element***
Windows	**Set**	**Access**	*insert: element*

```
has: element
        "Use sequential access to determine if element is in the set"
    [for: i from: 1 to: n do:
        [if: (element = (array lookup: i)) then: [return: true]].
    return: false]
```

Figure 1e. The user has selected the has: element method and its definition appears in the text pane.

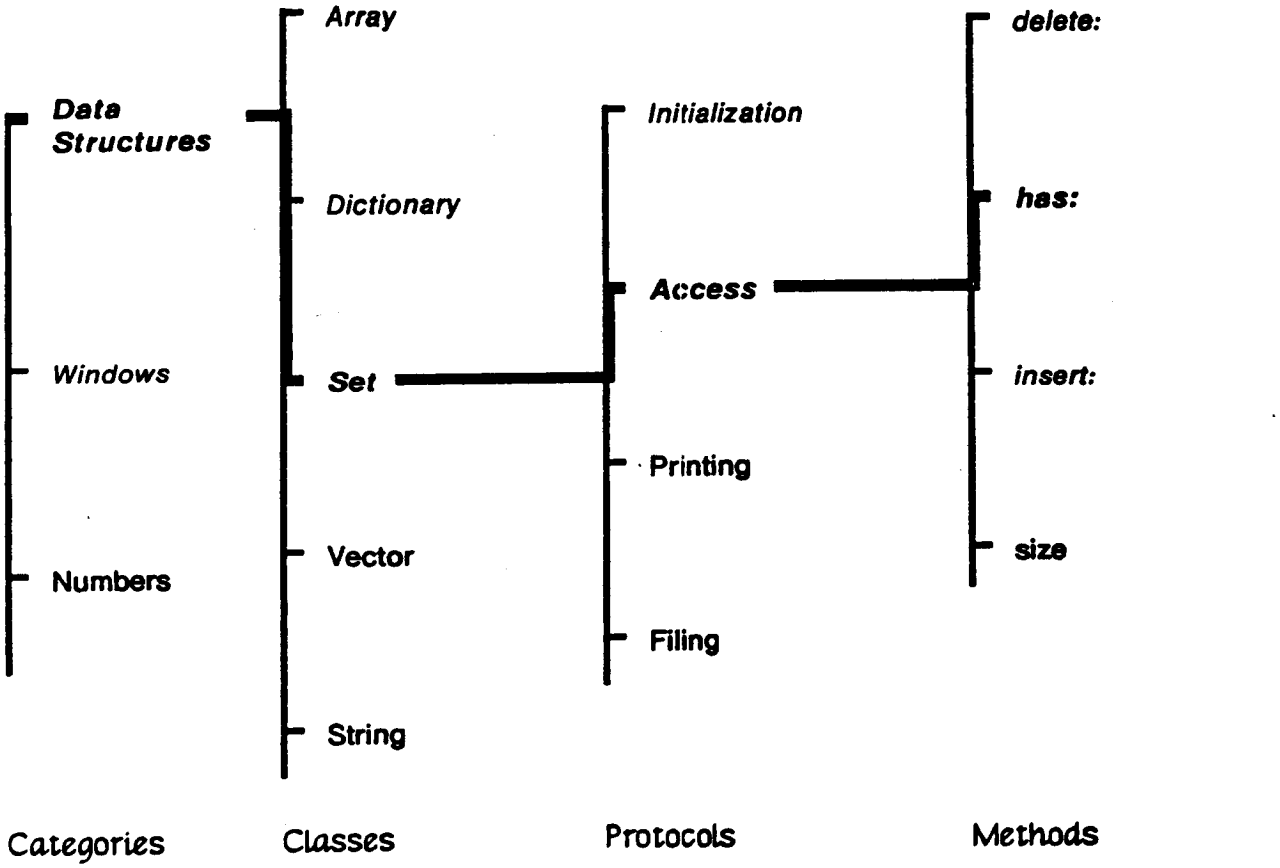

Fig. 2. A tree representation of the Smalltalk taxonomy. The path selected in the browser is shown in boldface. The slice of the taxonomy visible in the browser is shown in italics.

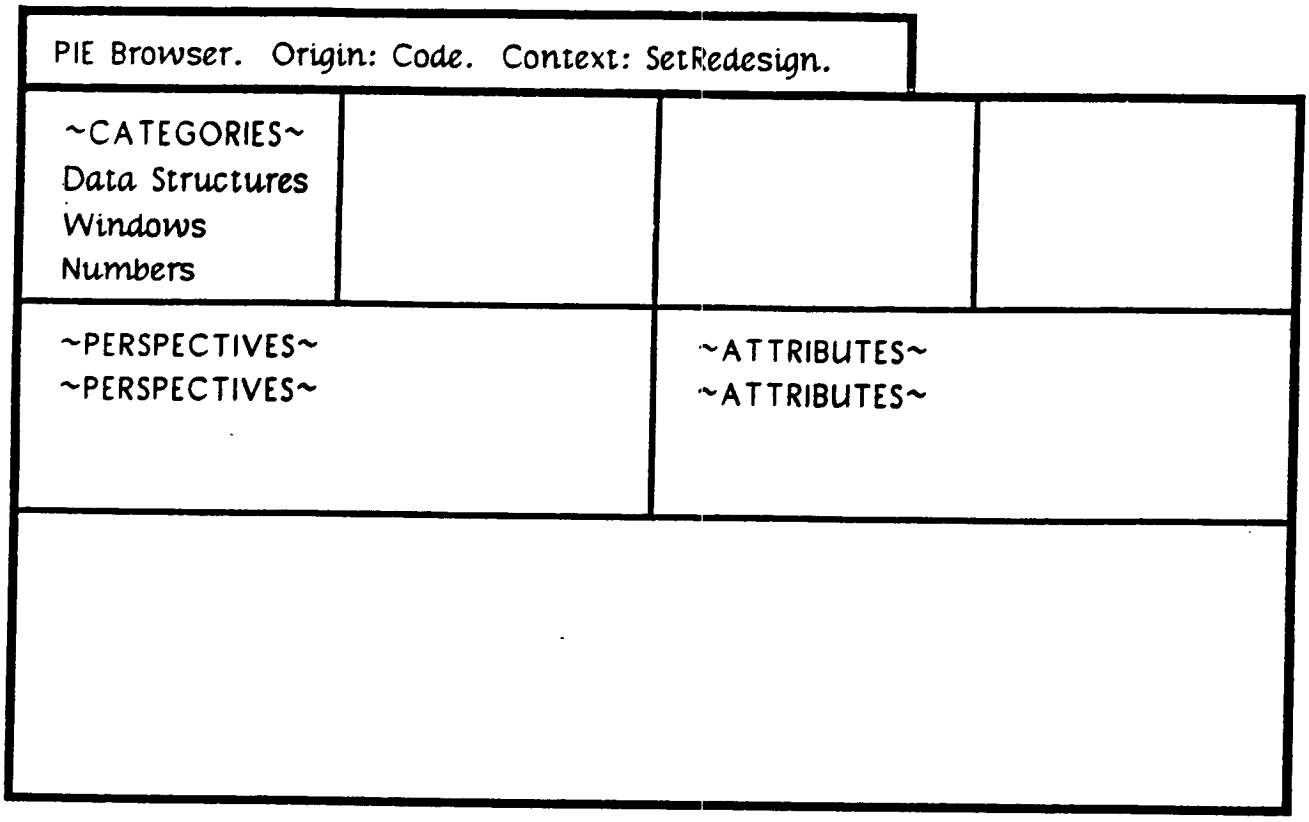

Fig. 3a. PIE browser viewing network with origin at Code.

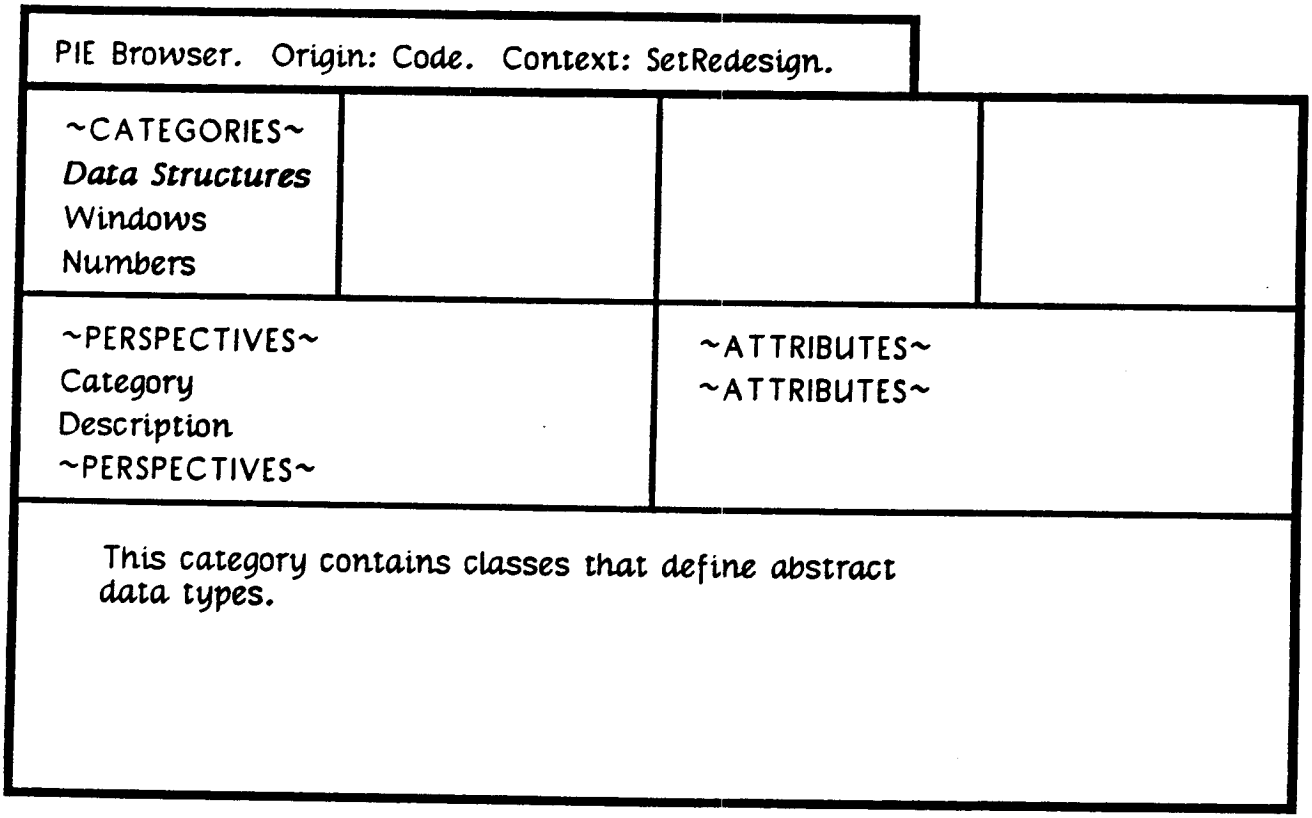

Fig. 3b. The Data Structure node is selected and its perspectives appear. The comment is the text attribute of the description perspective and is displayed by default.

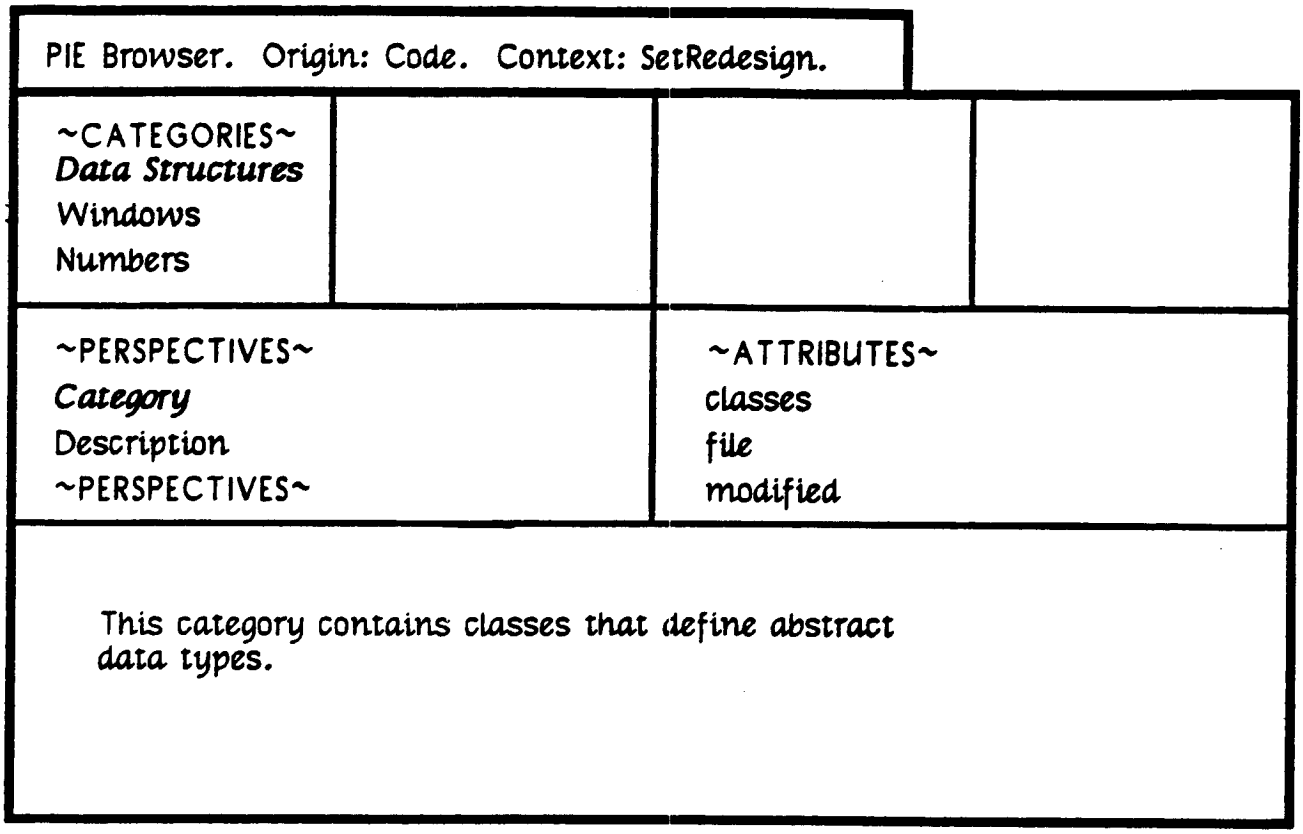

Fig. 3c. The category perspective is selected and its attributes appear.

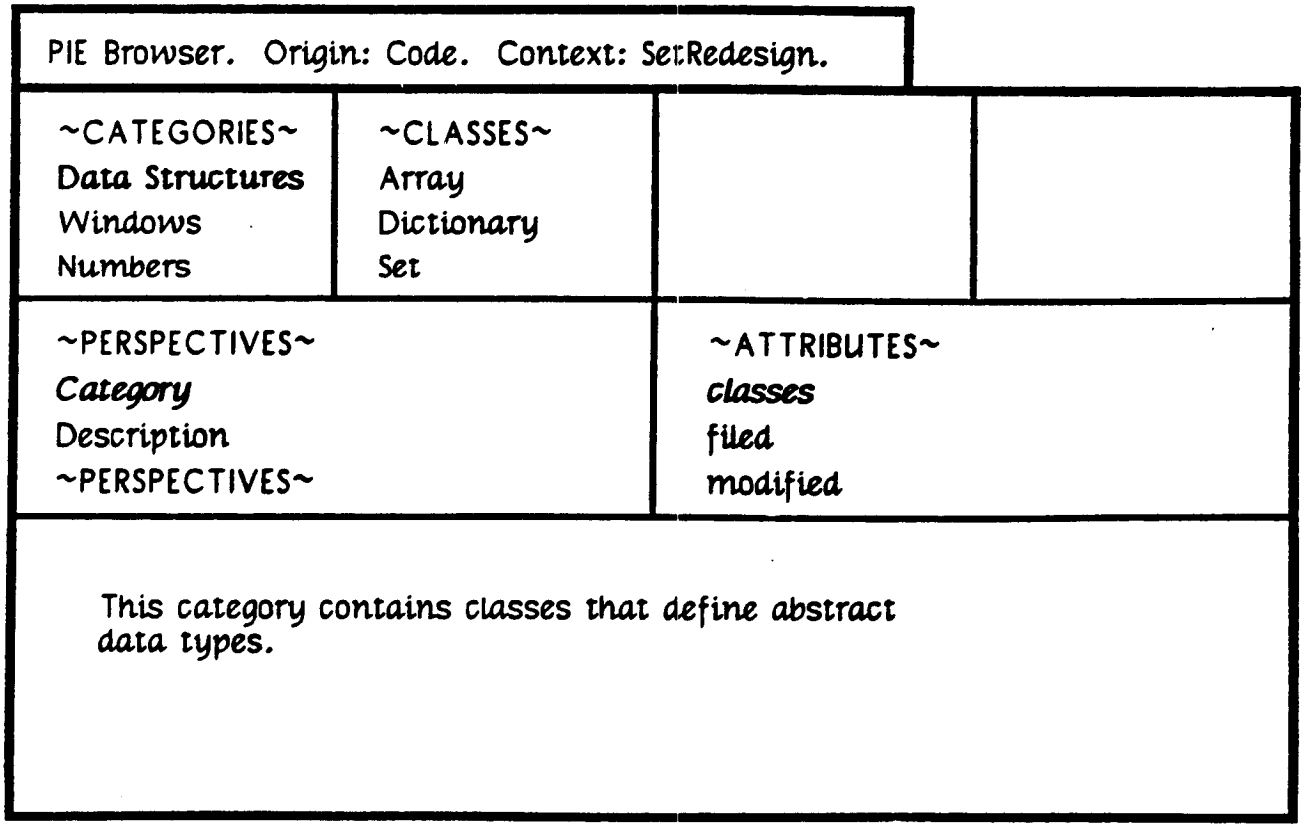

Fig. 3d. The classes attribute is selected and the list of classes appears.

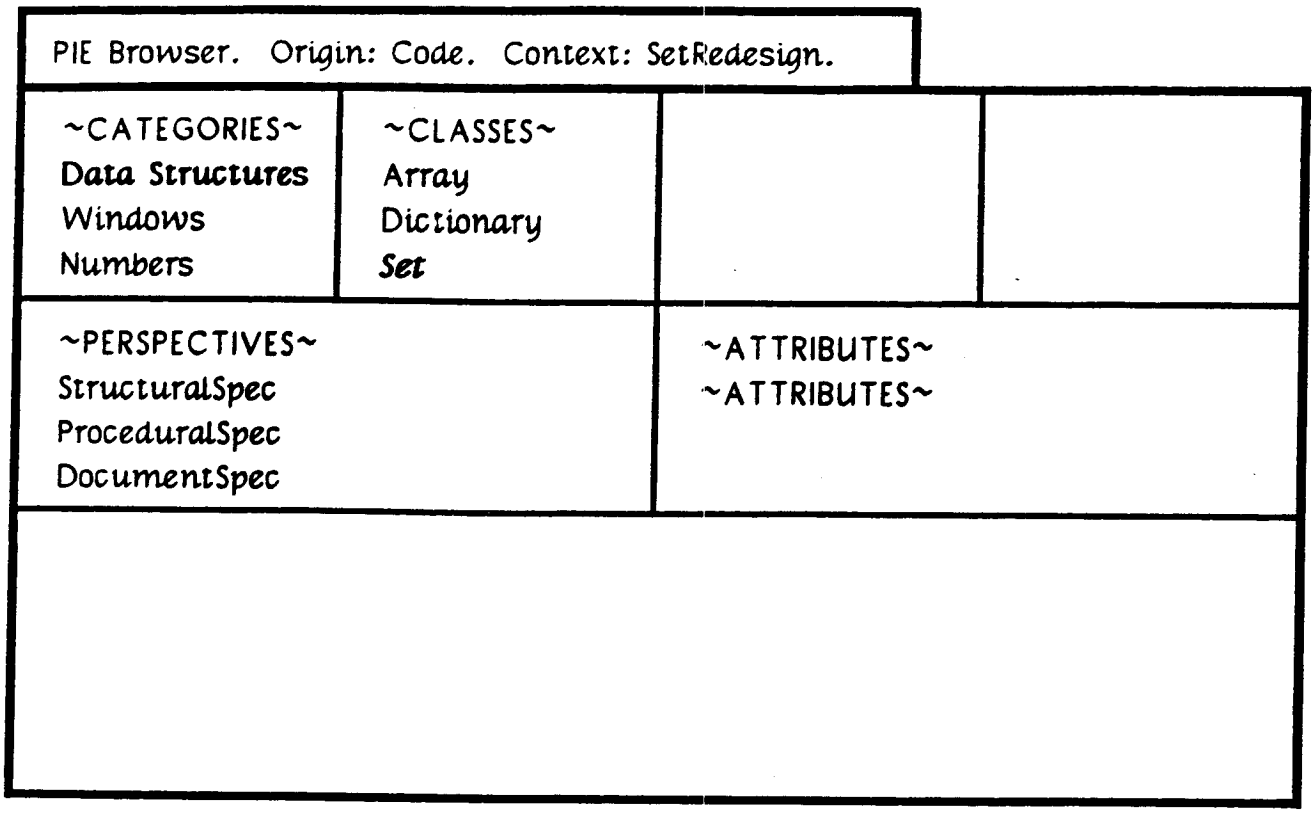

Fig. 3e. The Set node is selected and its perspectives appear.

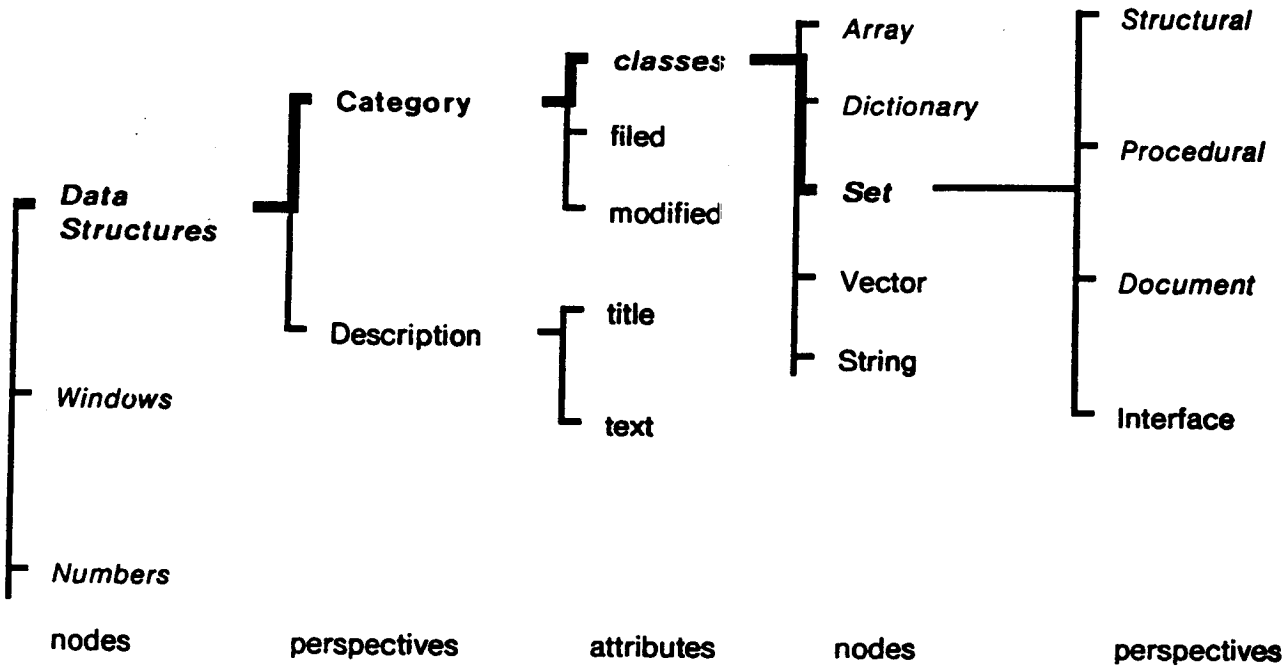

Fig. 4. A graphic representation of the PIE network. The path selected in the browser is shown in bold, the visible slice of the network in italics.

PIE Browser. Origin: Code. Context: SetRedesign.

~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Data Structures	Array	Initialization	delete: element
Windows	Dictionary	**Access**	***has: element***
Numbers	**Set**	Printing	insert: element

~PERSPECTIVES~	~ATTRIBUTES~
Method	***definition***
Description	message
~PERSPECTIVES~	contracts

```
has: element
        "Use sequential access to determine if element is in the set"
    [for: i from: 1 to: n do:
        [if: (element = (array lookup: i)) then: [return: true]].
    return: false]
```

Fig. 5a. The user is four levels deep in the PIE network.

PIE Browser. Origin: Data Structures. Context: SetRedesign.

~CLASSES~	~PROTOCOLS~	~METHODS~	~CONTRACTS~
Array	Initialization	delete: element	representation
Dictionary	**Access**	***has: element***	initialization
Set	Printing	insert: element	~CONTRACTS~

~PERSPECTIVES~	~ATTRIBUTES~
Method	definition
Description	message
~PERSPECTIVES~	***contracts***

Fig. 5b. The origin has been shifted to the Data Structures node, allowing the user to view the network one level deeper.