A Layered Approach to Software Design

by Ira P. Goldstein and Daniel G. Bobrow

CSL-80-5 December 1980

© Xerox Corporation 1980

Abstract: Software engineers create alternative designs for their programs, develop these designs to various degrees, compare their properties, then choose among them. Yet most software environments do not allow alternative definitions of procedures to exist simultaneously. It is our hypothesis that an explicit representation for alternative designs can substantially improve a programmer's ability to develop software. To support this hypothesis, we have implemented an experimental Personal Information Environment (PIE) that has been employed to create alternative software designs, examine their properties, then choose one as the production version. PIE is based on the use of layered networks. Software systems are described in networks; alternatives are separated by being described in different layers. We also demonstrate that this approach has additional benefits as a data structure for supporting cooperative design among team members and as a basis for integrating the development of code with its associated documentation.

CR Categories: 4.0 4.43 3.73

Key words and phrases: Design environments, source code control, program maintenance

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

Contents

1. Introduction 1

    1.1 The inadequacies of files
    1.2 Source code control systems
    1.3 Layered networks
    1.4 Previous research in Artificial Intelligence

2. A design scenario 5

    2.1 Smalltalk's implementation of class Set
    2.2 A layered redesign of class Set
    2.3 Redesigning the I/O behavior of Sets
    2.4 Representing contexts
    2.5 Composite contexts
    2.6 Communicating contexts
    2.7 Combining contexts
    2.8 Integrating software and documentation
    2.9 Complex designs

3. The user interface 19

    3.1 The Smalltalk browser
    3.2 The PIE browser
    3.3 Alternative interfaces
    3.4 Self-description

4. Remote storage 24

    4.1 Identifying existing nodes during the loading process
    4.2 Layer immutability
    4.3 Retrieving public contexts

5. Implementation 27

    5.1 Class structure
    5.2 Performance
    5.3 Memory management

6. Conclusions 30

7. Bibliography 31

1. Introduction

Most computing environments use files to express alternative designs. 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. In this paper, we argue that this use of files provides an inadequate structure for representing alternatives. We propose a notion of *layered networks* as a more suitable structure for representing an evolving design (and as an improvement over existing version control systems such as SCCS [Rochkind75] that represent software development in terms of changes to lines of source code). Our proposal is based on experience with an experimental system and our analysis of the deficiencies of present source code control systems.

We store software designs in networks whose nodes represent the modules, procedures and other entites of the design, and whose links represent relationships among them. Relationships are asserted relative to a layer. As an example, Figure 1 shows a block diagram of a simple software system and Figure 2 shows its network representation. All of the links in Figure 1 belong to layer A. Figure 3 shows a redesign involving changes to the definitions of procedures P1 and Q1. The new definitions are asserted in layer B whose links are shown dashed. The redesign has altered only the definitions of the procedures and not their comments, declarations or module membership. Hence, layer B contains no links regarding these attributes.

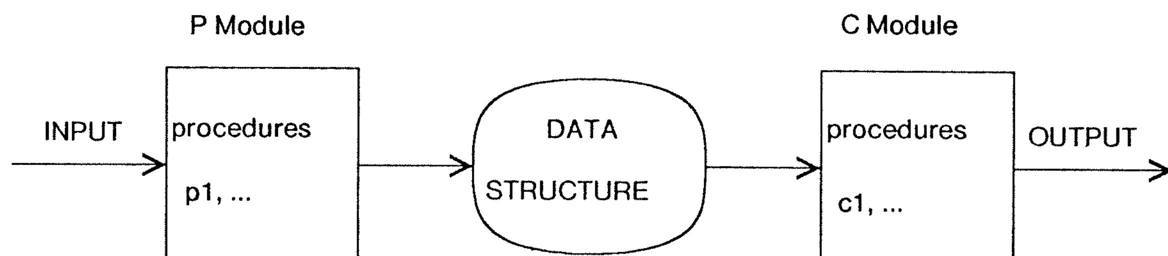


Figure 1. A block diagram of the PC system

Retrieval is performed with respect to a sequence of layers termed a *context*: the first value found for each link is returned. The new design is returned with respect to contexts in which layer B dominates layer A. The old design is returned for contexts containing only layer A.

1.1. The inadequacies of files

To exemplify the inadequacies of the traditional use of files to represent an evolving software system, consider how the development of the PC system would interact with the storage of its code in a standard file system. Suppose the source code for the two modules of the initial design is stored in files P and C. If a programmer develops an alternative design that requires changes to both modules, how can he store this alternative? Typically, he would create files P′ and C′ containing the new definitions plus any unaltered code. The result is that shared structure is stored redundantly. If subsequent development leads to modifications to procedures common to both alternatives, then these modifications must be made in both files. The need for redundant editing becomes progressively worse as the number of alternatives grows.

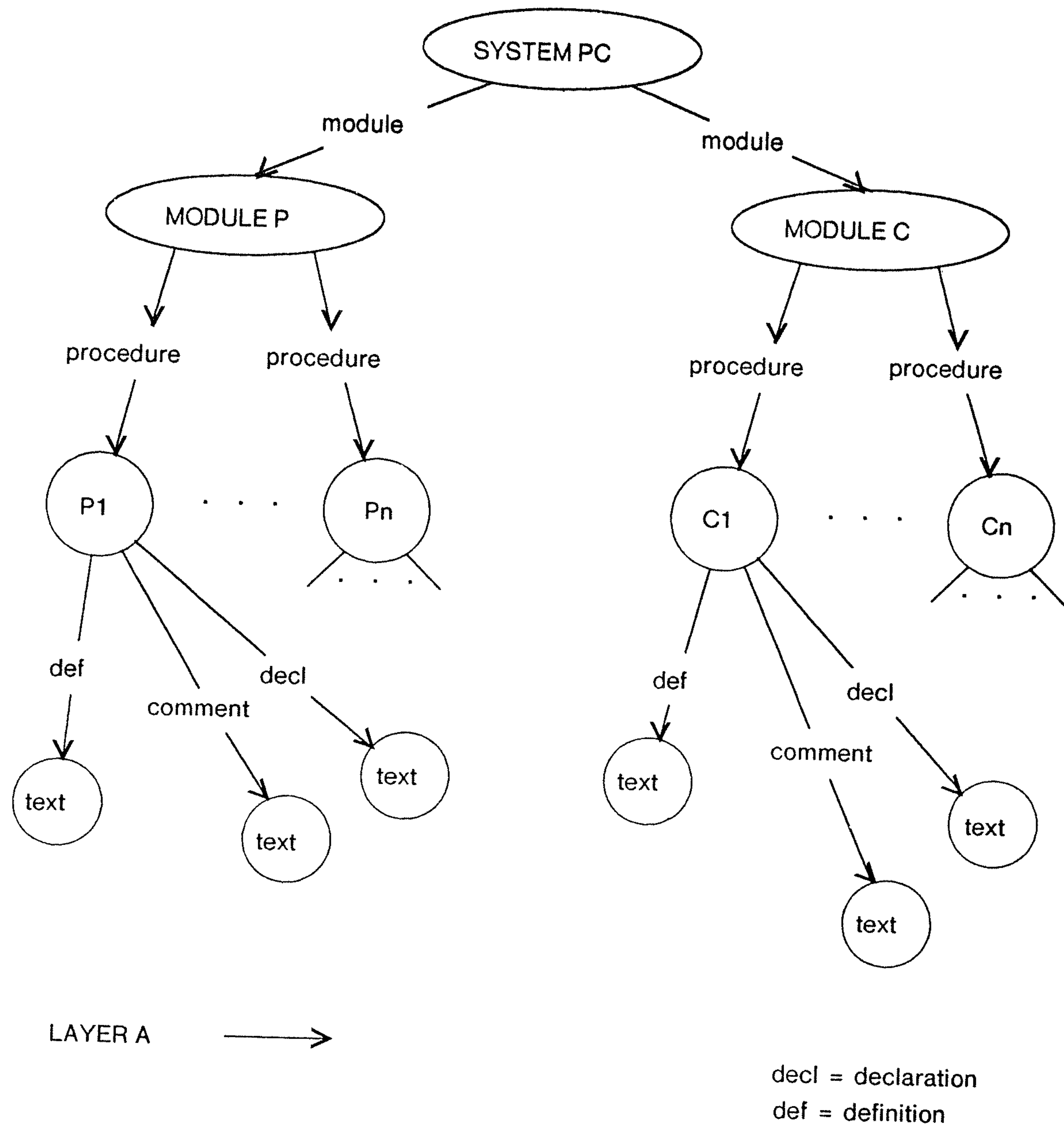


Figure 2. A network representation of the PC system

How else might the programmer store his design if he wishes to avoid the need for redundant editing? One option is to place the common code into a separate file. Altering a given procedure common to more than one design would then take place in only one place. The cost of this storage strategy is that files would no longer serve to group related procedures of a design. Thus, to obtain design flexibility, the programmer would give up an equally important feature—modular representation.

Another option is to use conditional compilation statements in the original source code and avoid the need for multiple files and the associated redundant storage. However, the problem of examining the set of changes common to a redesign is now complicated by the distribution of these changes across many files.

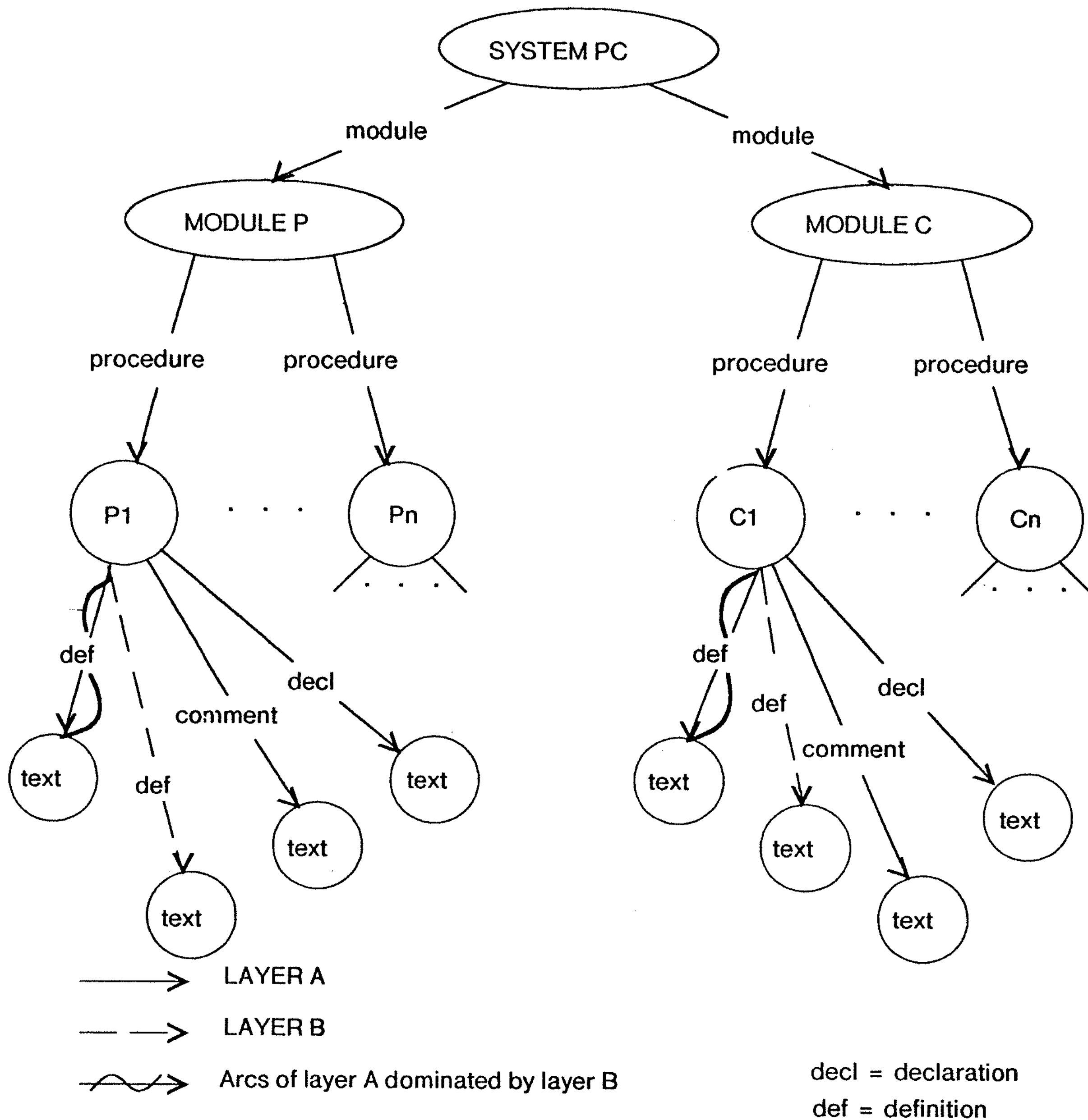


Figure 3. A layered representation of a design change

A related problem is coordinating change. The programmer must maintain descriptions of how various alternative designs are distributed among files. Sometimes this is done by adopting conventions in naming files, as we have implicitly done in naming the files for our second design P′ and C′. But such conventions are an impoverished means to describe a configuration and fail as the space of alternatives grows in complexity. Explicit descriptions of such configurations that do not depend on naming conventions are preferable.

Finally, there is the problem that designs represented as configurations of files are not reflected in the operational software. The result is that it is cumbersome to examine the structure and performance of a design interactively within the system. Switching from one design to another requires reading and writing the appropriate files.

Thus, the traditional use of files is unsuitable for representing alternative designs for three reasons. (1) Files are inflexible. Their utility to store modular parts of a design must be sacrificed to avoid redundant storage of shared structure. (2) File names are impoverished as a vehicle to encode the intended coordination between different files as part of a common design. (3) The representation of alternatives is not integrated into the running software environment.

1.2. Source code control systems

To remedy the first two of these deficiencies, editors have been developed that store changes of source code to a base file. One such system extensively used is the *Source Code Control System* [Rochkind75; Glasser78] developed as part of the Programmer's Workbench for the UNIX system [Ivie77]. In SCCS, changes are stored in terms of lines deleted and inserted in a file called a *delta*. A version of a file is computed from a base with the deltas applied in sequence. Changes do not have to be made in linear sequence; one can have branches in the chain representing alternative developments. A delta can be forced to apply to more than the version it was derived from. A delta also stores the name of the person who made the change, and the reason that it was made. The success of this system in practice—it has been used to control over 3 million lines of code by 500 programmers—makes it clear that having some mechanism for dealing with a finer granularity than files is important.

SCCS has the virtue that shared structured is not stored redundantly. A delta contains only changes. The description of a particular version of a system is specified by a delta sequence. The explanation for the differences between this version and earlier ones is stored in the comments associated with the new delta files.

1.3. Layered networks

Layers are similar to delta files. Information is not stored redundantly and versions are described by sequences of coordinated changes. However, our approach extends SCCS in several respects:

(1) We record changes relative to a network rather than a textual description of source code. The network provides a more structured description of software than text. A change to the first 10 lines in a listing has no intrinsic tie to the objects actually manipulated by the programming environment. Changing a link representing a relationship between two software objects does have such a tie. We have taken advantage of this tie to integrate the layer system into the software environment of the host language. Multiple definitions of a procedure can coexist in the environment and be-edited and tested side-by-side. The capabilities of an experimental programming environment based on layered networks are described in Section 2.

(2) We have developed a display interface for supporting a layered design environment. The Programmer's Workbench does not yet have this kind of interface for examining and manipulating code. The user interface is important since a representation for the evolution of designs is of little use if a user cannot manipulate it comfortably. Our primary technique for simplifying the presentation and manipulation of layered networks has been to use the network itself to record knowledge about defaults and constraints, so as to allow the system to assume the initiative in making certain common decisions.

(3) We have developed a program librarian described in section 4 that takes advantage of the network machinery to respond to retrieval requests for particular software. Layers and contexts are themselves described in the network, allowing the search requests to include requests for particular designs.

1.4. Previous research in Artificial Intelligence

Various kinds of layered databases have been explored in artificial intelligence research as a mechanism for representing alternative world views. (See, for example, Rulifson71; Hewitt71; SussmanMcDermott72; Hendrix 75; Cohen75). Generally the need to represent alternatives has arisen in planning programs. For example, a robot is analyzing alternative paths to reach some specified location. The terms *contexts* and *layers* are drawn directly from CONNIVER [SussmanMcDermott72].

Our application of a layered database differs from previous AI research 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. This has required that special attention be paid to the user interface. (2) Previous applications have demanded a uniform overhead in space and time for adopting the context machinery. Using the layered database was an all or nothing proposition in CONNIVER. We are exploring implementations 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 from restricted domains and of limited complexity. We have married the layer machinery to the host computing environment in such a fashion that any programming problem can be explored.

2. A design scenario

PIE is an experimental Personal Information Environment that employs layered networks to manage software development for any project undertaken in its host environment, Smalltalk [Ingalls76, Kay74]. Smalltalk is an object-oriented language that extends the notion of class and instance found in Simula [Birtwistle73]. This section presents a design scenario that we conducted using PIE to improve Smalltalk's implementation of the abstract datatype for sets.

In choosing a scenario, we faced two difficulties. The first was whether to use an actual Smalltalk example or convert to a hypothetical exercise in a more common programming language. We chose to use Smalltalk to illustrate the actual functionality of the system. However, PIE is largely independent of its host language. It is a system for building descriptions of software and contains few commitments regarding the language's interpreter or other characteristics. Thus, the techniques employed are readily mapped to other programming language environments.

Our second difficulty was choosing how complex a scenario to present. A simple scenario would not illustrate the utility of layers to aid complex design problems, but a complex scenario would be confusing in its own right. To resolve this dilemma, we have chosen a software problem that is simple quantitatively with respect to the number of design changes made, but more complex qualitatively in terms of the different kinds of changes

made to the implementation. Our redesign of the implementation for sets will include changes both to the efficiency of the implementation and to its input/output behavior. We consider combining designs, adjudicating differences, describing our design decisions in the network along with the modified software, and coordinating the whole project with associated documentation. A discussion of the application of layers to more complex design problems concludes this section.

2.1. Smalltalk's implementation of class Set

Abstract datatypes such as sets are implemented by *classes* in Smalltalk. A class defines a group of procedures, called *methods* in Smalltalk, and a set of variables on which they operate. Each method is invoked by a message pattern. Some of these messages are private; others are public messages that the class expects to receive from clients. Particular sets are *instances* of the class, i.e. each instance has some specified assignments of values to the *instance* variables of the class. Below is a simplified listing of class Set. Message patterns are shown in boldface and their methods appear below and indented. The listing is incomplete: for example, the definition of the method for deleting elements from the set is not shown.

```
Class new title: 'Set' instanceVariables: 'array n'

"Class Set employs an array with a position pointer n to represent sets. The
objects of the set are stored in the array from position 1 to n."

Initialization Protocol
  init
    "This method is conventionally executed when a new instance of class
    Set is created. It initializes the instance variables. The array variable is
    set to an array of size 8 and n is set to 0."
    [array ← Array init: 8.
     n ← 0.]

Public Protocol
  has: element
    "Testing whether an element belongs to the set is accomplished by
    iterating through the first n items in the array, checking for equality."
    [for⦂ i from: 1 to: n do⦂
          [if⦂ element = (array lookup: i) then⦂ [return: true]].
     return: false]
  insert: element
    "A new element is added to the set if it is not already present."
    [if⦂ (self has: element) then⦂ [return: false] else⦂ [self add: element]]

Private Protocol
  add: element
```

```
"A new element is added by loading it into position n+1 of the array
and incrementing the pointer. The array is copied into a larger array if
its free positions are exhausted."
[if: n=(array length) then: [array ← array growby: 10].
 array insert: (n←n+1) with: element.]
```

Printing Protocol

print

```
"This method prints a set by printing the string 'a set'."
['a set' print.]
```

2.2. A layered redesign of class Set

The first design goal we consider is to improve retrieval time by having the implementation convert from a sequential to a hashtable representation when the cardinality of the set exceeds some bound. The rationale for this redesign is that sequential access is less expensive in storage space and retrieval time when the set is small, but is not economic when the set is large.

We begin by generating in PIE a description of the current implementation. PIE is able to generate a network describing any class in Smalltalk from the internal Smalltalk representation for the class. This network is stored as a collection of instances of class Node, a class we created to implement the behavior of a network database. The network generated from the initial implementation is stored in a new layer—say layer A. The layer is placed in a new context which we shall name the *hashing context.*

The next step in the design process is to define the changes to the present implementation. A layer is created and added to the context to store these changes. This is layer B in Figure 4. Competing assertions in this layer will dominate those in layer A. Competing assertions are shown by links originating from the same point on the circumference of the circle representing the node. Links of layer B that only augment properties asserted in layer A such as the addition of another state variable (e.g. **limit**) are shown by links with the same label but these links do not originate from a common point on the circumference of the circle representing the node.

Below is part of the PIE-generated listing of the redesigned class with respect to the *hashing context.* Only parts of the public and private protocols are shown and comments have been removed. PIE has been instructed to **highlight** new assertions, derived from layer B, by printing them in boldface. The listing shows a new type of variable in the class definition. **limit** is a *class variable.* The value of a class variable is available to all instances. **limit** is used to specify the size at which the internal representation switches from sequential to hashed.

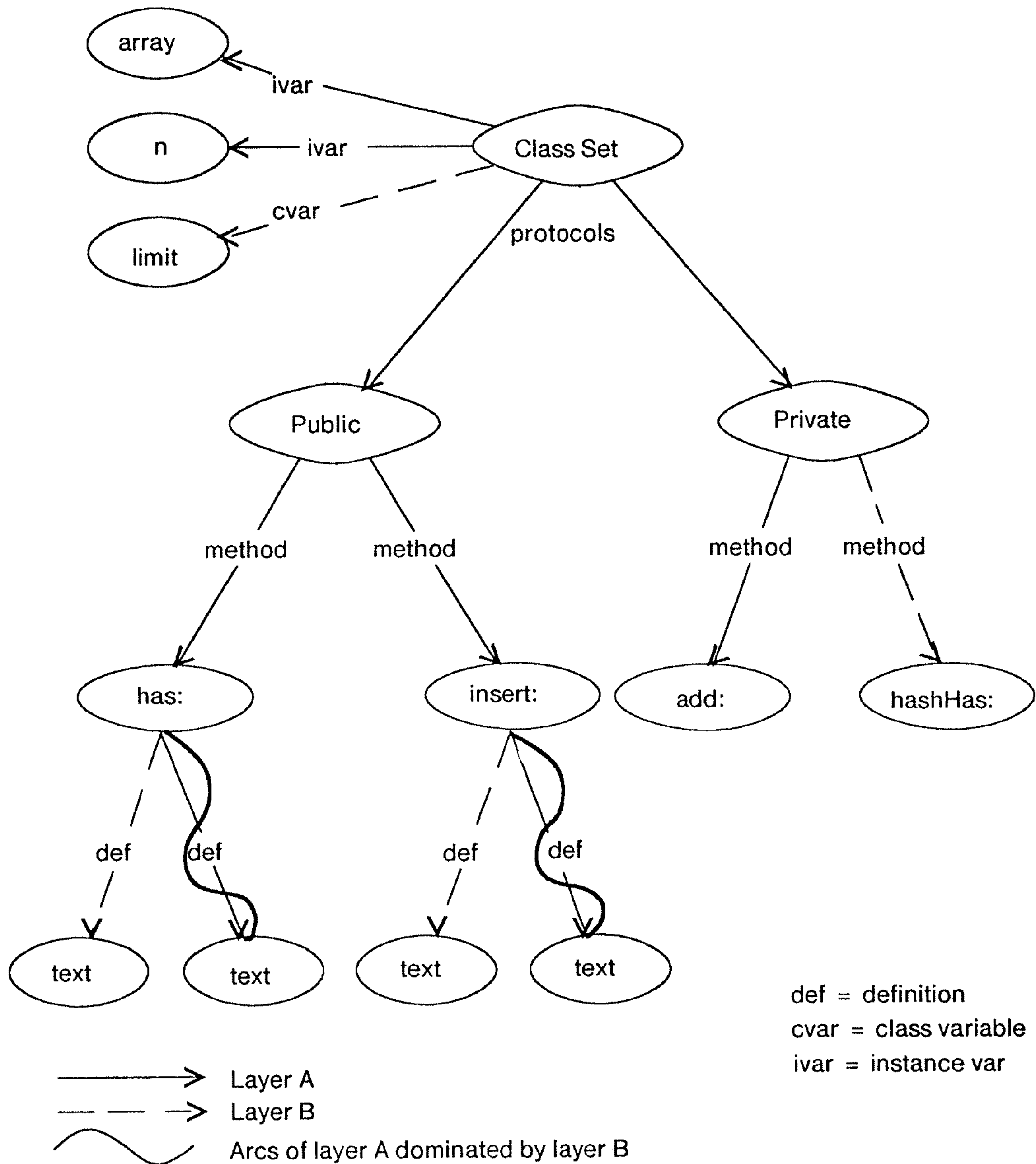


Figure 4. A partial view of a layered network representation for class Set. Layer A describes the original design; Layer B contains modifications. The Hashing Context contains both these layers with layer B dominating layer A.

```
Class new title: 'Set' instanceVariables: 'array n'
   classVariables: 'limit'
Public Protocol
   has: element
      [if: n<limit
       then: [for: i from: 1 to: n do:
                    [if: element = (array lookup: i) then: [return: true]].
              return: false]
       else: [return: (self hashHas: element)].
   insert: element
      [if: (self has: element) then: [return: false].
       if: n=limit then: [self convertFromSequentialToHash].
       if: n<limit then: [self add: element]
         else: [self hashAdd: element].]
Private Protocol
   add: element
      [if: n=(array length) then: [array ← array growby: 10].
       array insert: (n←n+1) with: element.]
   hashAdd: element
      ...
   hashHas: element
      ...
```

The user can test his design by *installing* the hashing context. Installation causes the Smalltalk interpreter to employ the definitions asserted in the specified context. Design changes are not immediately installed. This prevents premature modification of the underpinnings of the system before a design is complete. Hence, PIE maintains a distinction between the description context which is edited, and the execution context.

If further debugging is needed following installation, the programmer can create a new layer to store the changes to his design, then reinstall the context with this layer dominating the old layers. By placing the edits of each debugging session in a separate layer, the programmer can undo a set of changes that have proved unsatisfactory by removing the layer from the context and reinstalling.

Design exercises rarely consist of a single iteration through the design/debugging loop. In conducting this design exercise, a number of additional layers were added to the *hashing context*. For example, Layer C was added to correct the inconsistent treatment of the **limit** value. **has: element** treats **limit** as the lower bound of the hash representation while **insert: element** treats limit as the upper bound of the sequential representation. Layer C debugged the **has: element** procedure to use a $\leq$ test for comparing the size of the set and the **limit** value.

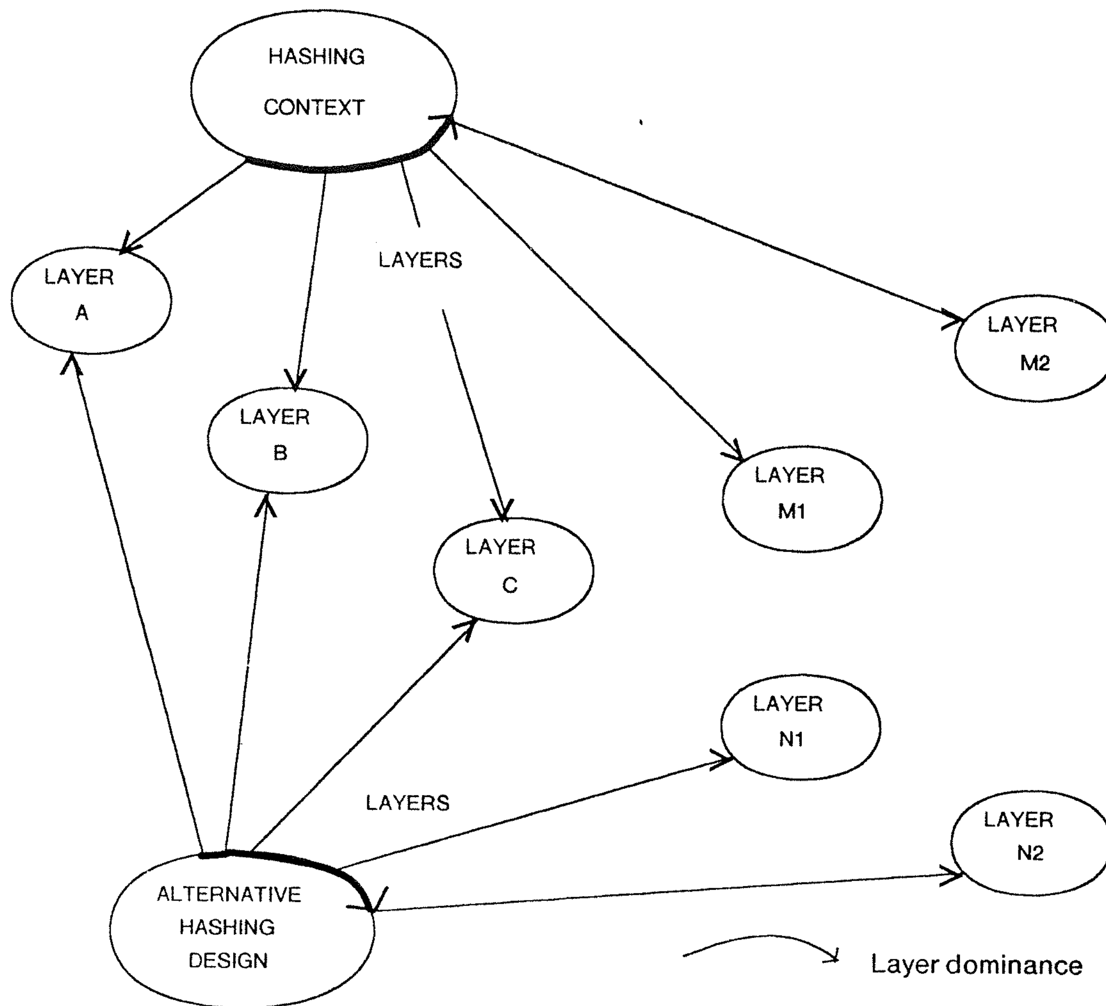


Figure 5. Alternative designs for hashing.
The dominance arrow points from least to most dominant layer.

Layers facilitate the comparison of alternatives*. For example, we considered different implementations of hashing in the redesign of class Set. Such analyses produced a network of layers as illustrated in Figure 5. The **m** and **n** layer sequences represent alternative designs.

---

* Easy switching facilitated our obtaining comparative performance measures for the original linear design and our mixed linear and hashing design for implementing sets. The parameters which determine performance of a set implementation are: the number of elements in a set; the ratio of membership tests to insertion and deletion; and the proportion of such tests which return false. With two tests for every insertion, and 33% returns of false, the choice of 7 for **limit** allowed the mixed design to dominate; with five tests for every insertion, **limit** should be 4 to provide overall better performance for the mixed design.

2.3. Redesigning the I/O behavior of Sets

To illustrate the ability of layers to manage interacting designs, we continue our scenario by pursuing a second design goal. This goal is to improve the I/O behavior of sets. Specifically, the goal is for an instance of class Set to print showing its elements enclosed in braces, e.g. as {A,B}, if the size of the set is less than some bound. Presently all instances of class set simply print as 'a set'. This redesign requires that we modify the printing method of class Set.

Before we make this change, we must decide where to store it. Since this is an independent modification of the code, our philosophy requires that we store these changes in a new layer, D. To make it easier to test and adopt the printing changes independently of the hashing changes, we create a new context, to be called the *printing context*, for these changes.

This context begins from the initial design of class Set and, therefore, its first layer should be the same as the first layer of the hashing context. If we examine or install this context, we get an implementation of class Set that only has the improved printing behavior. Layer D is added to the printing context to store the changes involved in this redesign. Below is the new method for printing sets stored in layer D.

```
print
   [if: (self size)<4
   then: ['{' print.
          for: i from: 1 to: (self size) do: [(self element: i) print].
          '}' print.]
   else: ['a set' print.] ]
```

This redesign becomes more interesting if we decide to include a modification to the Smalltalk reader that allows the string printed to be reread as a set. To accomplish this, we must modify the reader to recognize braces. We could put the required changes in layer D. Changes recorded in a layer can span module boundaries. But since the reader changes are independent of the altered printing behavior, it is better practice to put this set of modifications in a separate layer, say layer E. We can therefore test the two parts of the design separately, i.e. we can first install layer D to examine the printing behavior, then install layer E to examine the reader.

Layer E could be placed in an entirely separate context, but since we presumably want to adopt both the changes to the reader and to the printing procedure, it makes sense to include this layer in the set printing context. However, since the alterations are modularly stored in a layer, we leave open the option of creating a separate context to store changes to the reader that includes layer E.

The printing context now contains layers that make a coordinated set of changes to more than one module of the system: in this case, both the reader and a particular abstract datatype. This is not an unusual situation—despite a modular design, some modifications inevitably cross module boundaries, since the modularity is based on a particular partitioning of the design space, and such partitionings are not unique.

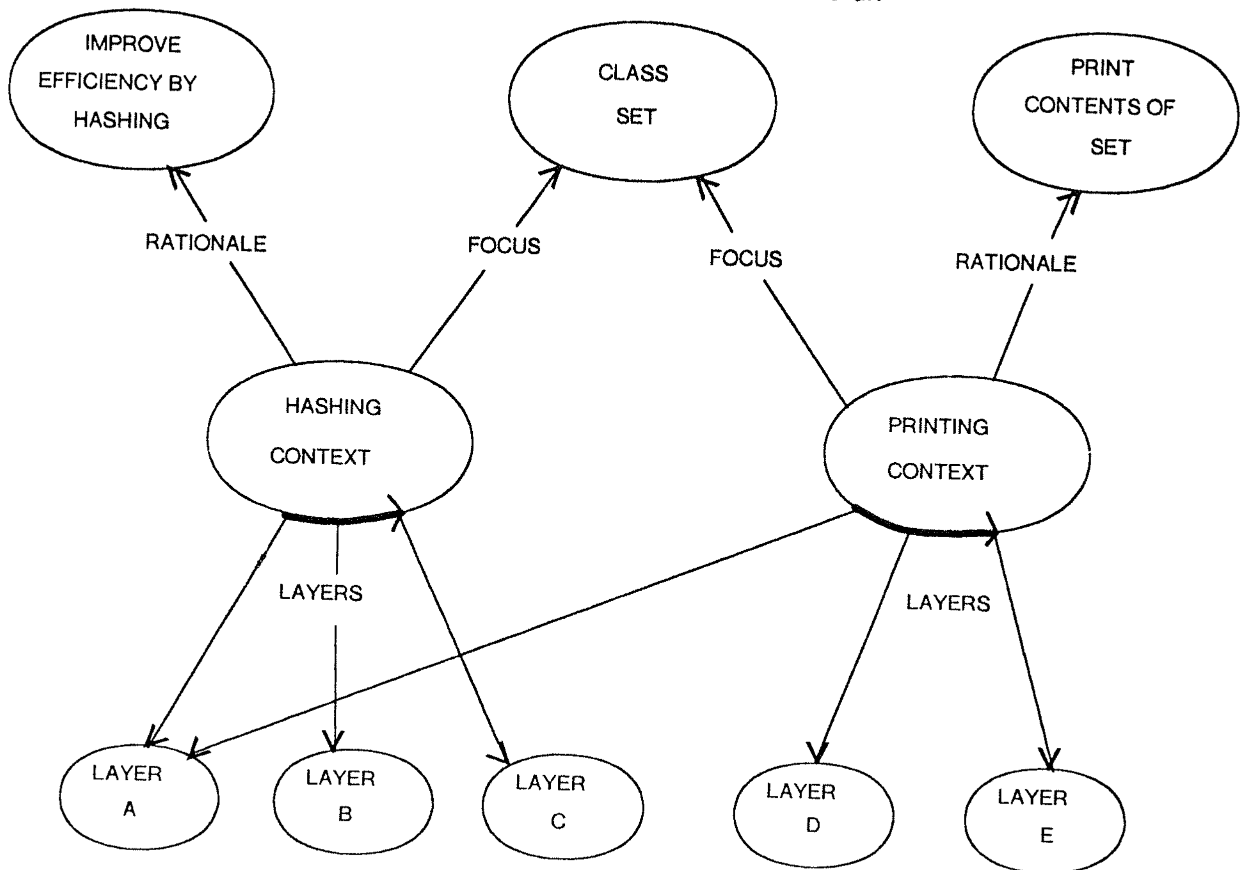


Figure 6. A partial view of the network description of layer and context nodes

2.4. Representing contexts

Layers and contexts are described by PIE in the same network as software is described. Figure 6 shows part of the network representing the hashing and printing contexts. The **rationale** attribute links a layer or context to a textual description of its purpose and the **focus** attribute points to the major classes being modified by the design. The **layers** attribute of a context node points to a sequence of nodes representing the layers.

Describing layers and contexts in the network has two important advantages. First, the user can search for a layer or context using the general network matching machinery provided by PIE. A search is initiated by specifying a description of some node in terms of constraints on the values of its attributes. Thus, a user can search for a node representing a context whose **focus** is class Set and whose **rationale** includes the subString *hashing*. The network description escapes the limitation of file systems in which the name of a file is burdened with the description of the file. Second, the user can manipulate layers and contexts using the same network operations used to manipulate code—i.e. the addition, deletion or modification of the attributes of nodes.

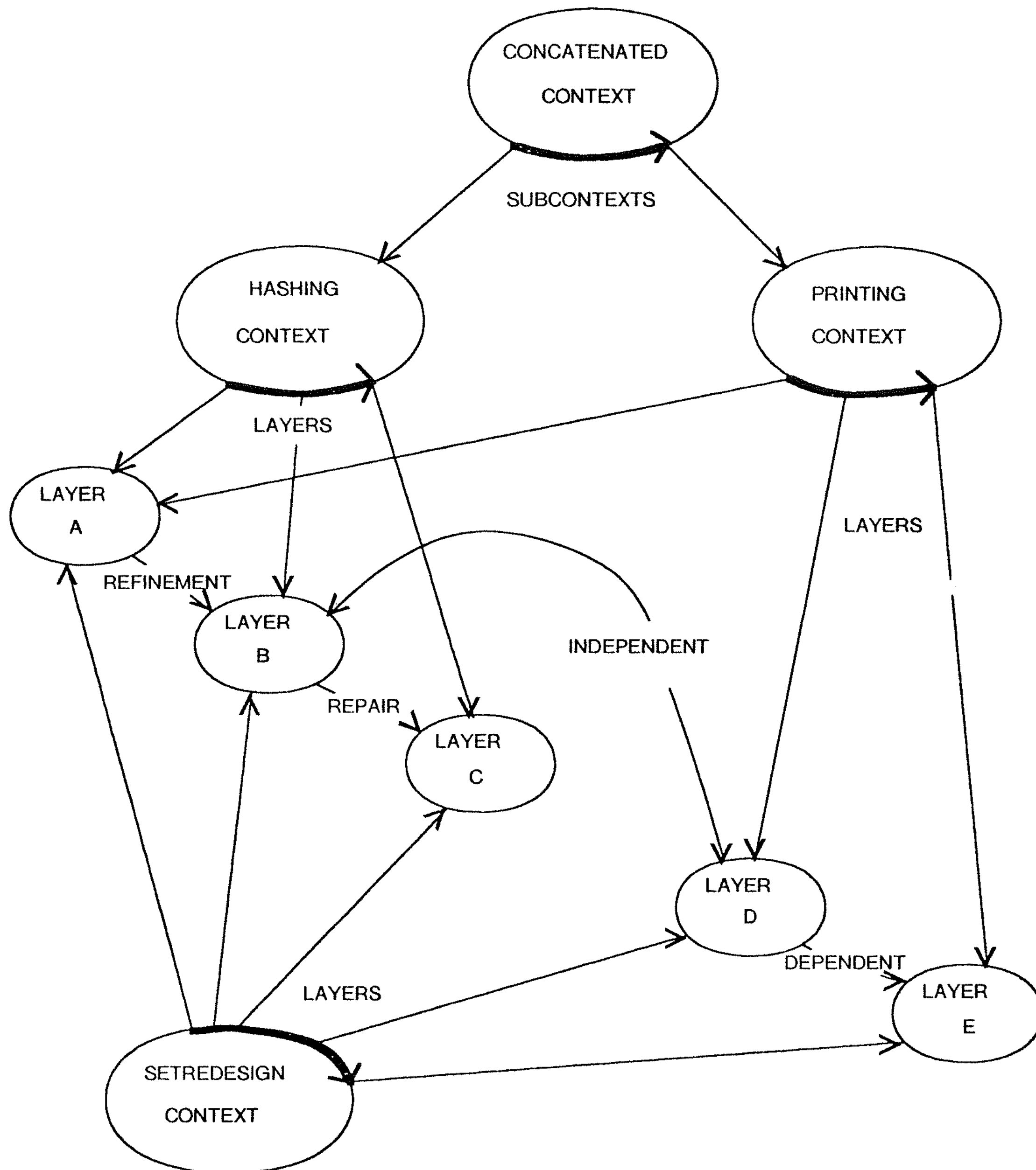


Figure 7. Two different kinds of composite contexts

2.5. Composite contexts

After we have debugged our two redesigns of class Set, we will want to combine them. We can do this by creating a composite context built from the layers of the existing contexts. This is the *SetRedesign* context shown in Figure 7. We have not simply concatenated the layers of the hashing and printing contexts. This would produce the sequence: A, B, C, A, D, E. The second occurrence of layer A would inadvertently dominate the changes in layers B and C.

We may wish to impose the constraint that if new layers are added to the hashing or printing contexts, then they are automatically included in the composite context. This can be accomplished by a capability that PIE provides for defining procedures that are attached to nodes. These procedures are triggered by adding or deleting attributes of the node. By defining such a procedure and attaching it to the **layers** attribute of the hashing and printing contexts, we can have it synchronize these contexts with the composite context.

Concatenating new layers of the printing context to the hashing context is justified by the independence of the printing and hashing refinements. This is therefore a useful comment to include in the network. Figure 7 illustrates various design comments. Layers B and D are commented as independent refinements, layers D and E as dependent refinements and layer C as a repair for layer B. When layers are combined into new contexts, these comments are checked and the user is alerted to questionable combinations such as adding a layer without its subsequent repairs. This description also allows other programmers to examine the network of contexts and layers and understand their relationships.

Interactions between design decisions can lead to conflicting values in two layers. In SCCS, conflicts are noticed only to the extent that two modifications touch the same line of code. In PIE, the granularity of detected overlap is at the level of the method; if two layers have changed the same method, then there is a potential conflict. In addition, we have in PIE a mechanism for explicitly expressing dependencies among a number of methods. This allows us to find some potential interactions when there is no structural overlap.

To resolve conflicts between layers, a new layer could be added to the composite context that resolved any differences in design decisions. This layer would only apply to the composite design and not to the individual designs since it would not be included in their contexts.

Finally, we could have created a composite context. The *concatenated* context in Figure 7 is such a context. PIE treats the layer sequence of such a context as the concatenation of the layer sequences of its sub-contexts. This combination strategy is appropriate when the constituent contexts are independent. But here, the common use of layer A makes it inappropriate.

2.6. Communicating contexts

We have discussed this design project so far from the standpoint of an individual programmer. But many projects are collaborative. Layers facilitate cooperative design by supporting on-line interaction that is analogous to two programmers scribbling on a common listing. One programmer can transmit to another programmer a set of changes to the first programmer's design by sending a layer. PIE supplies comparison functions for identifying the changes between layers. Below is a PIE-generated listing of class Set with respect to the redesign context plus a layer transmitted by a collaborator. The collaborator's layer redefines the **has: element** procedure and includes an annotation stating the rationale for the change.

```
Class new title: 'Set' instanceVariables: 'array n'
   classVariables: 'limit'
Public Functions
   has: element
      [if: (self size)≤limit
       then: [return: (self seqHas: element)]
       else: [return: (self hashHas: element)].
```

*Annotation from Danny received 4/1/80, 3:15pm: **I think that a more subroutinized definition will pay off in the long run.***

The advantages of this level of intimacy in communication would not ordinarily be sufficient to offset the disadvantages that might arise from an unwanted intrusion by the sender into the recipient's workspace. A programmer might be justifiably relucant to load a file from a collaborator directly into his workspace. Despite the appeal of being able to examine the new software with design tools in the software environment, the software may contain modifications that overlap and interfere with software developed by the programmer.

Layers avoid this problem. The sender s modifications are contained in a separate layer. Hence, they can be loaded without destroying changes stored in separate layers. The user can install then, examine their performance, then undo them if desired simply by deleting the layer from his context.

2.7. Combining contexts.

Following the receipt of contributions from a collaborator, the need arises to select some of these proposals and combine them with the programmer's existing design. A design environment should make it easy to examine overlapping designs and to select pieces for combination into a joint design. A layered environment facilitates this since coordinated sets of change have been localized into layers. One needs a user interface for comparing layers.

PIE provides several interfaces for comparing layers and contexts. We have already seen the ability of the system to generate listings that highlight contributions from different layers using fonts and faces. A comparison of the differences between the two layers is accomplished using the interface shown in Figure 8. This interface presents the user with a display screen divided into three regions called *windows*. The upper window shows the (node, attribute) pairs on which two contexts or layers differ. The user can select a pair from this list. The interface then shows the differing values in the two contexts in the middle and lower display windows.

The interface creates a new layer, entitled the merge layer, in which the programmer can selectively copy those (node,attribute,value) triples that he wishes to include in his own context. If the merge layer is then added to his context, it will dominate his old layers and supply the new information.

Figure 8 shows the user comparing his redesign of class Set (stored in the hashing context) with improvements suggested by his collaborator. The user is presently examining the definition of the **has: element** procedure. His collaborator has suggested a more modular definition. The user is about to add this definition to the merge layer, thereby

Comparing SetRedesign context with DannysLayer

~Differences~

Set vars

has: annotation

has: definition

hashHas: element

. . .

~SetRedesignContext~

```
has: element
   [Ifg (self size) ≤ limit
    theng [forg i from: 1 to: n dog
               [ifg (element = array lookup: i) theng [return: true]].
             return: false]
     elseg [return: (self hashHas: element)] ]
```

~DannysLayer~

```
has: element
   [ ifg (self seqMode)
     theng [return: (self seqHas: element)]
     elseg [return: (self hashHas: element)] ]
```

Figure 8. The Set redesign context is being compared with a collaborator's suggestions. The user has selected the definition attribute of the has: method in the upper pane. The alternative values appear in the lower two panes.

adopting his collaborator's suggestion. A consequence of this is that the user must also assert in the merge layer definitions of the required subroutines: **seqHas: element** and **hashHas: element.**

Understanding such dependencies is facilitated by examining commentary supplied by the designers regarding the rationale of their choices. But this requires that commentary be coordinated 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 in the same layer as the one which records the change, thus keeping them coordinated.

For complex designs, the merge process is non-trivial. PIE does not eliminate this complexity. What it does provide is a more finely grained descriptive structure than files in which to manipulate the pieces of the design. It highlights differences explicitly, and provides a place to document dependencies, and the rationale for changes.

2.8. Integrating software and documentation

Program comments are ordinarily second class citizens. While the programs may have elaborate structure, the comments are only strings. A uniform network representation facilitates the integration of structured comments with the corresponding software. A node representing a class can point to the node documenting its behavior. Nodes representing procedures of the class can point directly to nodes representing the appropriate parts of the documentation. Comments that apply to more than one part of a software system can point to multiple nodes. The PIE-user interface described in the next section supports the integrated development and perusal of such network structures.

Figure 9 shows a partial view of the network documenting class Set, developed at the same time as the software. The description of the algorithms employed is linked to the procedures implementing these algorithms. A user can enter the network from any node and traverse it as his interests dictate. Hardcopy is generated using user-specified filters on how much of the network to print and on the ordering. Thus, a listing of the source code need not contain the entire discussion of the underlying algorithms—a programmer can choose how much detail to see.

An additional advantage of representing documentation in the network is that layers can be used to represent alternative drafts and coordinate joint authorship. Layers are, after all, a general means to organize an evolving design. An author can explore alternative organizations with layers that modify **section** links. Revisions of paragraphs can be stored by means of alternate **text** links. Comments regarding the rationale for a change can be recorded with **annotation** links and stored in the same layers as the change. Filtered views of the network can highlight or suppress such annotations.

Coauthorship can be facilitated by adopting the convention that a coauthor place his revisions in separate layers. Additional advantages can be obtained if the authors use separate layers for recording different kinds of change: for example, separating spelling corrections or minor style improvements from major revisions or suggestions for altering the content of the document. This separation allows an author to accept without further examination layers from a coauthor containing minor revisions, thereby freeing time to

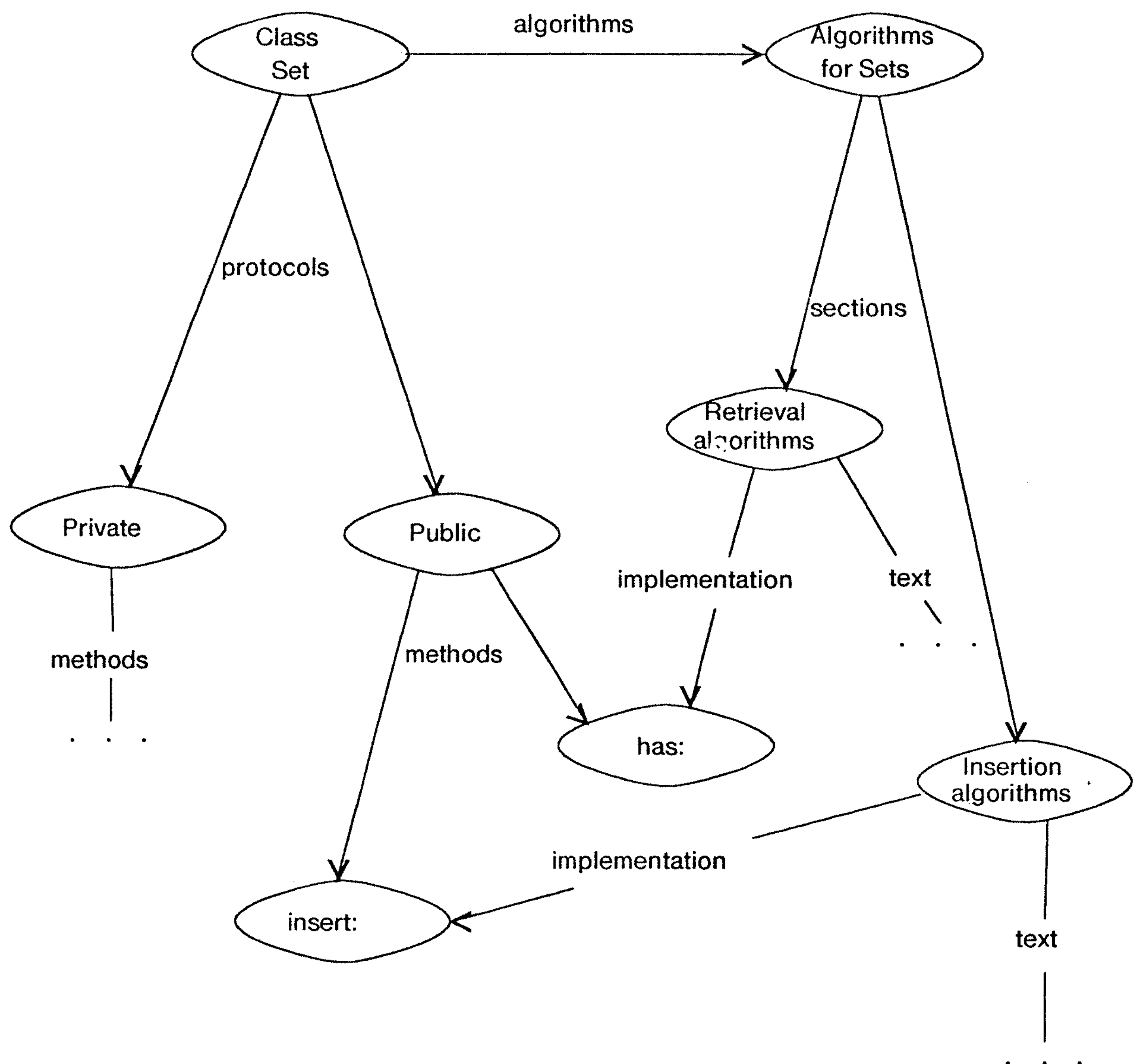


Figure 9. A partial view of a network in which nodes representing code and documentation are inter-linked.

concentrate attention on other layers. The purpose of the layer can, of course, be recorded in the network for examination both by other authors and by various formatting and comparison tools.

Across a period of time, we have built up such conventions regarding the purpose of layers, and have found them crucial for organizing the coauthorship process. We have also found that these conventions apply equally well to software: a layer can be correcting the grammar or style of a module, making minor revisions in its parts, or proposing a major structural change.

2.9. Complex designs

Massive redesigns can involve changes to hundreds of procedures distributed over dozens of modules. We believe that the machinery described here for layers and contexts provides capabilities that are suitable for such complex real-world software problems based on our analysis of the deficiencies of present source control systems.

As with systems like SCCS and unlike systems that store the entire source code for a package in a single file, we store a set of changes modularly with very little redundant information. This allows forks in designs to be explored and facilitates the creation of specialized configurations from selected subsets of the layers.

Unlike SCCS, we represent software and design configurations in a network that has important structural advantages over textual descriptions. First, the network supports a representation in which there is a natural locus for all of the properties of a given object including both source and compiled code—such properties would typically have to be distributed across a textual listing or even across different files. A designer can readily examine all of the properties of some object by interrogating the network. Second, a network allows us to move two ways across a link. Thus we can move from the nodes describing variables to their classes or vice versa. A designer can therefore traverse the network to explore some the consequences of a redesign. Third, the formalized descriptions of a network support search and matching operations to find desired objects of a design, including layers and contexts themselves. Fourth, the network facilitates the formation of various kinds of composite designs. Hierarchical file directories provide some aid of this kind for file-based source code systems, but are less powerful than the network architecture of PIE. Finally, the network strictly dominates text since a node can have a **source code** attribute that points at text describing the software.

3. The user interface

PIE's ability to represent alternative designs comes at the price of a more complex, context-sensitive representation. For this price to be affordable, the user interface must simplify the presentation and manipulation of the database. In this section, we discuss a user interface that has proved successful for expert users, and several alternative interfaces that may be more appropriate for novices. We also discuss the use of additional description provided in the network to specify reasonable default behavior for the interface.

3.1. The Smalltalk browser

The PIE interface is modelled on the standard Smalltalk interface for examining and altering code. The Smalltalk interface is shown in Figure 10 and is called a **browser**. It is a display window built from six sub-windows called *panes*. The top pane is the title pane. Below it is a row of four panes that display, from left to right, categories of classes, classes, protocols and methods. The lower pane displays text associated with the most recently selected item. The browser as shown is in a state arrived at by the following process. (1) The user selected the category ***Data Structures*** in the upper left pane. (Selected items are shown in boldface italic.) This selection caused the classes of this category to be displayed in the pane to the right labelled classes. (2) The user then selected the class ***Set***. This caused the protocols of this class to be displayed in the pane labelled protocols. (3) The user then selected the protocol ***Public***, causing the methods of this protocol to be displayed

Smalltalk Code Browser

~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Kernel Classes	Array	Initialization	delete: element
Data Structures	Dictionary	***Private***	***has: element***
Numbers	***Set***	Public	insert: element
Windows	Vector	~PROTOCOLS~	~METHODS~
. . .	. . .		

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 10. The Smalltalk Code Browser

in the pane labelled *methods*. (4) Finally, the user selected the method ***has: element*** causing its definition to appear in the lower pane. The user can now edit this definition using commands associated with the lower pane. Thus the browser allows the user to examine and change any method in the Smalltalk system. It has found wide acceptance in the Smalltalk community.

3.2. The PIE browser

PIE organizes the representation of Smalltalk code into a network, rather than a hierarchy. The network includes nodes that represent categories, classes, protocols and methods with links describing their hierarchical organization. However, the network includes non-hierarchical relationships. A node representing a method might be linked to more than one protocol in more than one class. Nodes representing code can point to nodes representing documentation and vice versa. Nodes representing methods can point to nodes that describe dependencies between the method definitions.

Figure 11 shows the generalization of the standard Smalltalk browser that we have implemented to examine this network. Moving from one level of structure to the next is a two step process. The first step is to select a node. This causes the names of its attributes to be displayed in the pane immediately below. The lower pane is labelled with the type of the node. The second step is to select an attribute. The value of this attribute is then displayed in the upper pane to the right. The pane in which it is displayed is labelled with the attribute name. For example, when the user selects the node representing the **Data Structure** category, the names of the attributes of this node are shown below in the pane labelled *ACategory*. These attributes include the category's classes and documentation. The user must then select one of these attributes to see the next level of node structure. In Figure 11, the user selected the *classes* attribute. This caused the nodes representing the classes of the category to be displayed in the upper pane to the right under the label *classes*. The user then selected a path through the network that allowed him to reach the **has: element** node.

The definition of the **has: element** method shown in Figure 11 differs from that shown in Figure 10 since the PIE browser is viewing the network with respect to the SetRedesign context while the Smalltalk Browser was viewing the original Smalltalk specification for class Set. The context being employed for viewing the network is shown at the top of the browse window.

Multiple PIE browsers can be placed on the display screen simultaneously to view designs from different contexts. For example, a second browser could be created and associated with the original **SystemCode** context. This would allow the user to compare the two designs.

3.3. Alternative interfaces

We are experimenting with alternative interfaces that display the network in different ways. The goal is to minimize the difficulties new users encounter in mastering the network architecture. We are exploring browsers that display the network as a graph similar to that shown in Figures 3 and 4, as an outline, and as a nested set of boxes. Figure 12 shows these last two views of the new Set design. The **outline browser** employs a two dimensional layout editor developed by Bob Flegal and Diana Merry of the Xerox PARC Learning Research Group. The **box browser** is a descendant of a summer project done at Xerox PARC by Bill Finzer of San Francisco State University.

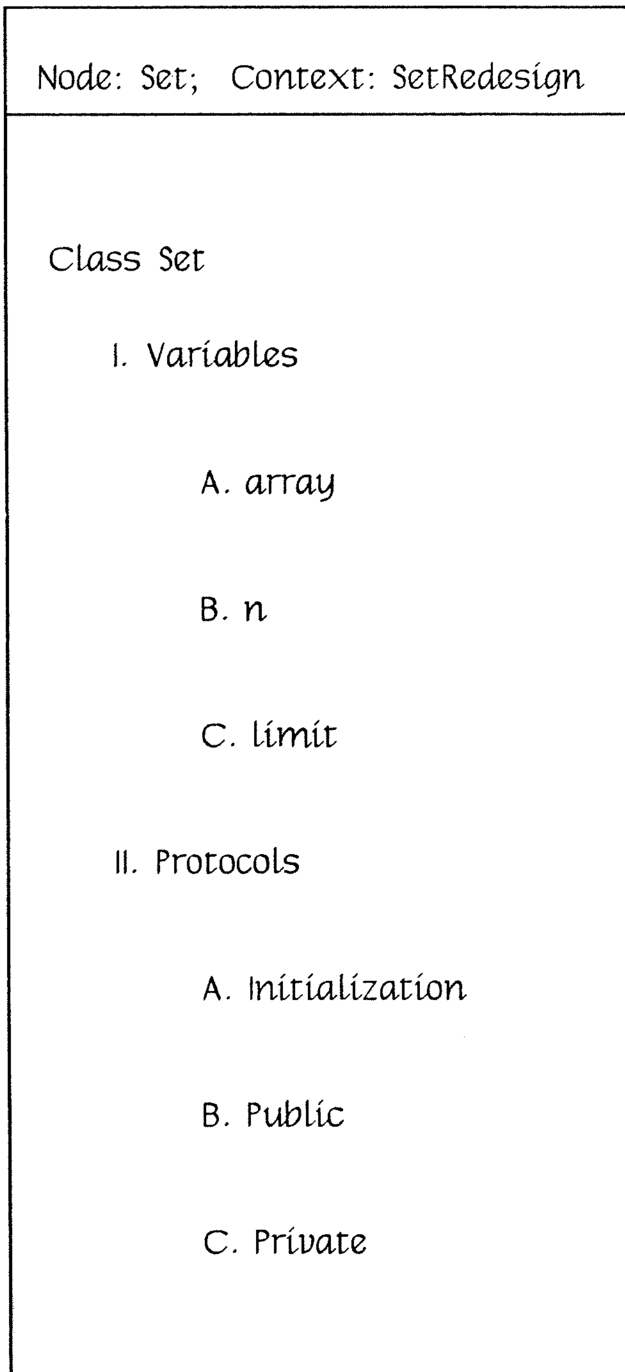
ORIGIN: Code ; Contexts: SetRedesign

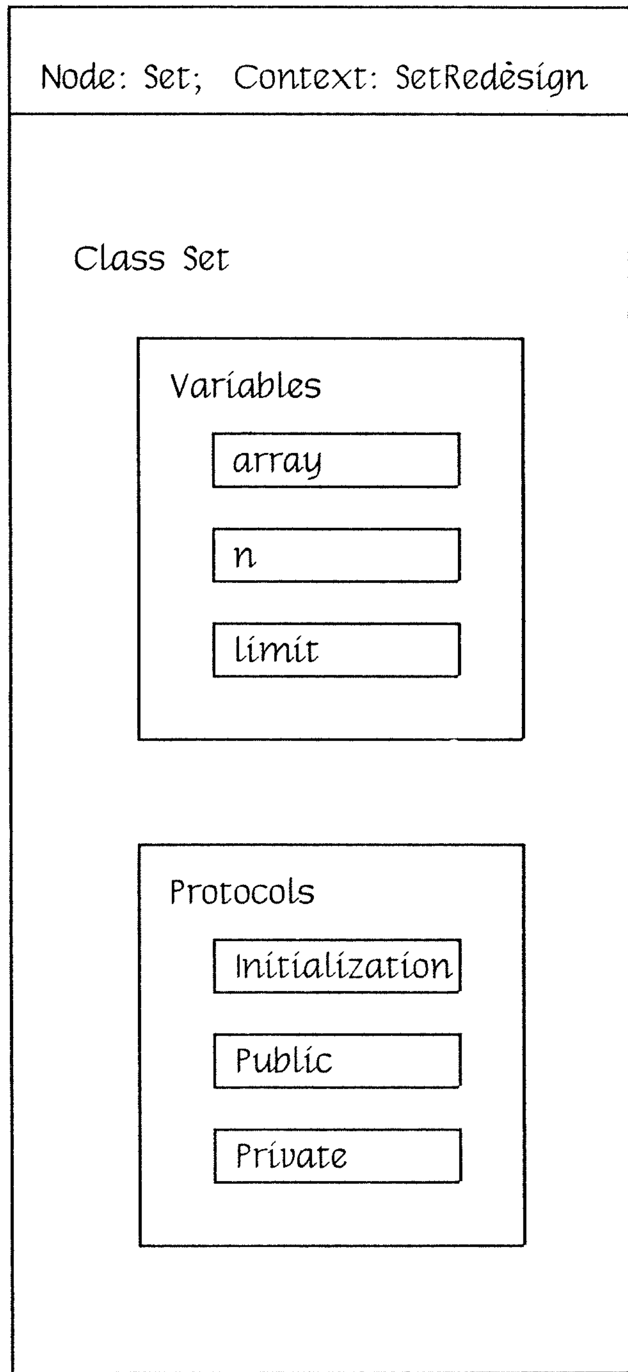
~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Kernel Classes	Array	Initialization	delete: element
Data Structures	Dictionary	Private	***has: element***
Numbers	***Set***	***Public***	insert: element
...	...	~PROTOCOLS~	~METHODS~
~ACATEGORY~	~ACLASS~	~APROTOCOL~	~AMETHOD~
classes	***protocols***	***methods***	***definition***
comment	comment	comment	comment
documentation	variables	variables	variables
...	...	...	...

~definition of has: node~

```
has: element
   "Use sequential access to vector if size ≤ limit, else use hashing."
   [ If: (self size) ≤ limit
     then: [for: i from: 1 to: n do:
                    [if: element = (array lookup: i) then: [return: true]].
            return: false]
      else: [return: (self hashHas: element)] ]
```

Figure 11. The PIE Browser viewing nodes from the SetRedesign context





THE OUTLINE BROWSER THE BOX BROWSER

Figure 12. Alternative browsers for examining the network. Any item appearing in either browser can be expanded to reveal its substructure.

We have not yet had sufficient experience with different users to know which interface is the most congenial for unravelling the complexity of context-dependent network designs. Perhaps more than one is appropriate, depending on the user's purpose—e.g. obtaining an overview, examining a part of the design in detail, making a coordinated set of changes. Our present plan is to develop a variety of interfaces and analyze their acceptance by various users.

3.4. Self-description

One problem common to all of these browsers is that the number of choices needed to move from one node to the next is larger than that required by the simpler hierarchy of the original Smalltalk. When a code object is selected in Smalltalk, there is no ambiguity regarding the subordinate objects to be displayed. This is reflected in the fact that the labels of the browser panes are fixed. When a PIE node is selected, however, there is a potential ambiguity regarding which subordinate nodes to be displayed: the selected node may have more than one attribute that points to other nodes. The user must specify the desired attribute. Furthermore, the appropriate context must be chosen for viewing the value of the attribute. Thus, the richer data structure of PIE comes at the price of investing more effort to traverse the information space.

We minimize the effort required by the user by supplying default specifications of expected decisions. These defaults are supplied by means of additional description associated with the node being examined. For example, this description includes a specification of the default attribute and context. As a result, the browser can make an informed guess regarding the desired information to be displayed. By an appropriate choice of defaults, the PIE browser can mimic the behavior of the standard Smalltalk browser. The default attribute of a category is its classes, of a class its protocols, etc. The default context is generally specified in the self-description associated with the class node, since designs generally span entire classes. The user can override defaults by explicit selection. Therefore, the user expends additional effort only when traversing non-standard paths.

We have found that this knowledge-based behavior of the browser is critical to reducing the overhead in employing the system. Selecting the appropriate context and attribute is tedious. Forgetting to select the appropriate context is disastrous. Controlling the interface with meta knowledge stored in the network both improves efficiency and reliability.

4. Remote storage

We have not yet discussed how contexts and layers are stored remotely for backup purposes and to provide access by other programmers. In an early implementation, layers and contexts were stored as files in the following way: The layer file contained assertions regarding the contents of the layer, i.e. the (node, attribute, value) triples and was named by a unique identifier associated with the layer. The context file contained the names (identifiers) of layers owned by the context. Thus, loading a design was a multi-step process: the user first examined a file directory for the context with the desired name, retrieved its list of layer identifiers, then loaded these layers from a second file directory.

There are several subtleties and some deficiencies in this scheme. First, how does the loading process identify that the an assertion in the filed layer is referring to an existing

node? The filed layer does not know the local address of the node and node titles are not unique. Second, how do we deal with changes to layers, since we do not allow multiple versions of a layer? Third, how can we avoid reducing retrieval of filed contexts to examining a list of file names? This seems to return us to the limitations of file names that layers and contexts were intended to eliminate. This section discusses each of these points in turn.

4.1. Identifying existing nodes during the loading process

Our method for identifying existing nodes is to assign each node a unique identifier when it is first created. This identifier is unique across all users. It is generated from the time of creation and the serial number of the creating machine, and defines a community wide address space. When a layer is written onto remote storage, relations between nodes are described by the name of the relation and the identifiers of the participating nodes. When the layer is loaded into a new environment, these identifiers are checked against a dictionary of identifiers and existing nodes in the environment. If present, the identifier is replaced by a pointer to the existing node. If not, a new node is created and assigned this identifier. Thus, different PIE systems have their own copies of a node while a particular PIE system has at most only one copy of a node, regardless of the number of layers in which it is referenced.

Unique identifiers are a satisfactory basis for communication when one team member creates a set of nodes, then transmits his layers to another team member for subsequent development. The unique identifiers are included in this transmission. When the second team member returns his contribution, information can be attached to nodes created in the original space.

Unique identifiers provide no help when one team member wishes to attach some information to a node in the workspace of another team member whose unique identifier is not known. He may only know a description of the node, for example, that it represents the "retrieval function for the set module." This problem is treated by allowing the sender to transmit a search request to the recipient. The search request, if successful, returns the identifier of the desired node. Naturally the search will be unsuccessful if the sender formulates his request in terms not understood by the recipient's environment. However, this is generally mitigated by the common vocabularies that team members develop.

Unique identifiers do not deal with the problem of two team members independently creating separate nodes to describe the same object. To cope with this problem, PIE contains functions for comparing the descriptions of nodes. Based on these comparisons, the user must decide if an unintentional coreference has occurred and take appropriate action, such as using one of the nodes as the preferred one and deleting the other.

4.2. Layer immutability

We avoid multiple versions of a layer by adopting the convention that layers are immutable once they have been stored remotely and made available for public use. Change is limited to adding new layers and updating the context file. Thus, to obtain the latest release, a user reexamines the context file and loads any new layers. To make private modifications to a software design, a new layer is added to the context in which the changes are made, rather than altering the retrieved layers.

There are several advantages to immutability. First, it allows a user to judge if his release is current by comparing the layer identifiers retrieved from the context file with the identifiers of the layers have been loaded. It is not necessary to load fresh versions of those layers, since they do not change. Second, the immutability convention allows a PIE system to treat local memory as a cache. The local memory stores those layers currently being used. Trying to use a non-local layer causes a fault that initiates a load of that layer from the remote store. If space is unavailable, an existing layer is erased from local memory. The layer need not be rewritten on the remote store if it has been previously filed, since it cannot change.

Immutable layers work well for cooperative design in which updates are not required immediately. They are not appropriate for real time interactions as they do not solve traditional problems of deadlock and resource sharing. Design problems being attacked by a team, however, do not require this level of intimacy. Indeed, designers benefit from being able to examine proposed changes without having them be immediately inserted into their workspace.

4.3. Retrieving public contexts

We improved upon our early use of a file directory to store public contexts by providing a PIE system as a public context directory. We have previously discussed how PIE improves upon file directories for a user's local contexts. These contexts are stored in the network and a user can retrieve a context using PIE's search machinery. The network description avoids the limitations of file names. Hence, it was a natural choice to use PIE itself as a public directory.

To store or update a public context, a programmer transmits to the public PIE system a network description of the context. This network fragment contains the name of the context, the identifiers of its layers, and any attribute/value properties such as its rationale and focus that the creator chooses to make public.

The public PIE system can now service requests for a particular software package that range from the name of the context to a network description of its properties. Furthermore, if an ambiguity arises or if the user simply wishes to examine the available software, the public PIE system can support browsing through its network of contexts just as a private PIE system does.

An advantage of a PIE system serving as a public directory is that it can respond differently to different kinds of users. Some users may be designers who want the complete history of a design while other users may only want the completed system. For the former, PIE can deliver the layer identifiers. The programmer can then read these layers into his workspace, and compare their contents with the contents of his own layers. For the latter, PIE can deliver the Smalltalk source code using its standard installation machinery for converting a layered design into running software. In this case, the user need have only a kernel Smalltalk. No PIE machinery is necessary. Thus, the cost of a machine powerful enough to support flexible design need only be borne by those engaged in design. For other members of the environment, a kernel system is sufficient.

The machinery to employ PIE as public context directory is implemented, but we have not tested it on a large user community. We have successfully employed PIE in this way on an experimental basis for collaborative software projects undertaken by the authors.

5. Implementation

In this section, we discuss the Smalltalk class structure of the PIE system, its performance, and two implementations of layers.

5.1. Class structure

PIE's classes are grouped into three categories: **display**, **database** and **semantics.** The **display** category implements the browser shown in Figure 11. In implementing the browser, we took advantage of previously existing Smalltalk classes which provided most of the desired behavior. The PIE browser is built from four classes. The four panes in the top row that display lists of nodes have identical behavior and are instances of one class, the four panes in the middle row that display attributes are instances of a second, and the bottom pane employed for text editing is an instance of a third. A fourth class coordinates the behavior of the individual panes. These four classes are subclasses of existing display classes in Smalltalk and they inherit much of their behavior from their superclasses. Where there are differences, the PIE subclasses have methods that override the behavior of their superclasses. Thus, Smalltalk's hierarchical class structure made it straightforward to define a powerful, individualized display interface.

Multiple browsers can coexist on the screen. Each browser is a different set of instances of the display classes. This allows separate browsers to examine different regions of the network from different contexts. At any time, the user can create a browser whose initial view is centered on the current selection.

The **database** category contains classes that implement the basic network machinery. Instances of class Node are used for nodes in the network. Each node has a unique identifier, an optional pointer to a meta-node on which default information is stored, and a set of attributes pointing in a context-sensitive fashion to other nodes. Two mechanisms for implementing context-sensitive retrieval are described in Section 5.3.

The **semantics** category contains classes termed *perspectives* that are assigned to a node in order to assert that the node represents a particular kind of entity, such as a Smalltalk class or a text document. Perspectives are a type mechanism. Assigning an instance of some perspective class to a node supplies methods that implement the behavior for the type of object that the node represents. For example, perspectives representing Smalltalk code have methods for compiling themselves. The nodes for class Set, its protocols, and its methods all had the appropriate code perspectives assigned. Perspectives representing documents have methods for printing themselves in different formats. The PIE manual has such a perspective, and, hence, can be printed in various filtered formats—with or without footnotes, with or without annotations, with or without paragraph summaries.

It is possible for a node to be assigned more than one perspective when it is sensible to view the entity represented by the node from more than one point of view. One example of this occurs for nodes that represent documents. In addition to the document perspective, they may be assigned a bibliographic perspective. This second perspective supplies the appropriate formatting behavior when the node appears as part of a bibliography.

There is presently a library of 15 perspective classes available for users of PIE including perspectives for describing classes, protocols, methods, documents, user profiles, layers and contexts. However, this category is open-ended and intended to be expanded by the user.

5.2. Performance

PIE runs with excellent response time on a Dorado, a high speed micro-programmable personal computer that runs Smalltalk at approximately one million instructions/second [LampsonPier 80]. The response time is perceived in terms of the time to refresh the browser's panes following a new selection. This response time is satisfactory—a user perceives no delays.

The criticial limitation of the present Smalltalk implementation is the size of its virtual memory. Smalltalk-76 [Ingalls78] supports an address space of only 32,000 objects. This includes all of the objects defining Smalltalk itself. As a result, there is insufficient space to build large PIE networks. No more than 1,000 nodes can be in the local address space at any one time. Since there are approximately 3,000 methods in Smalltalk, it has not been possible to build a PIE network describing the entire Smalltalk system.

A new Smalltalk implementation soon to be available has a 30-bit virtual memory. Given this capacity, the possibility exists of transforming PIE from its present experimental status to a permanent part of Smalltalk's programming environment.

A PIE/Smalltalk marriage imposes no unavoidable costs on software developed under its auspices. PIE allows a user to trade flexibility for efficiency. Maximal efficiency can be obtained by employing 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 maintain a history, then he can convert to a node description of his Smalltalk code and develop his software in a context-sensitive fashion. From this point forward, the evolutionary history is maintained. The price is increased retrieval time and storage space. If these costs become prohibitive, the user can convert back to pure Smalltalk. A solution less extreme than converting to pure Smalltalk is to summarize the dominant assertions of a sequence of old layers into a single new layer, thereby reclaiming some space and speeding retrieval. The user can then continue in a context-sensitive fashion by placing subsequent assertions in a new layer. In either case, the user can store the old layers remotely before deleting them from his local workspace. This allows the user to recreate the history of his design if desired.

The implementation of PIE was a substantial design project in its own right. Hence, once the kernel PIE system reached a sufficient degree of reliability, we applied it to its own development in a number of ways. A major application was to explore the various display interfaces described earlier. Various display projects involved upwards of one hundred changes distributed over more than twenty classes. The layered approach proved particularly useful for exploring closely related designs for the interface since these projects involved a great deal of shared structure. Furthermore, since there was no *right* design, it was useful to preserve alternatives for a while in order to obtain experience with each of them. Finally, we adopted the convention of asserting changes to program documentation in the same layers as changes to the code. This greatly facilitated keeping the documentation consistent with a rapidly changing prototype.

5.3. Memory management

We have experimented with two implementations for layered networks. One stores the context-sensitive information in the node, the other in the layer. The node-centered storage is more consistent with the Smalltalk metaphor: each instance of a class knows all of the information regarding its state. The layer-centered design has better memory management behavior.

The node-centered design associates the attribute of a node with a dictionary of layers and values. In this design, retrieval of a value of an attribute **a** of node **n** with respect to a context is based on the following algorithm:

1. Retrieve the layer/value dictionary for this attribute stored in the node.
2. Sequentially lookup each layer of the context in this dictionary. If the layer is found in the dictionary, return the associated value. If no layer of the context is found in the dictionary, return "?".

The second design is layer-centered. Layers own a dictionary that associates (node,attribute) pairs with a value. The retrieval algorithm for the value of attribute **a** of node **n** with respect to a context is to sequentially lookup (n,a) in the dictionary of each layer in the context. If (n,a) is found, return the associated value. If no layer of the context stores (n,a,) return "?".

To understand the virtues of the layer-centered scheme, we must discuss how the network is stored externally. Layers constitute the unit of secondary storage—i.e., when a user completes a design, he writes the layers defining the design onto secondary store. If the design is subsequently examined, the user must read these layers into his main store. The layer-centered data structure of (node,attribute,value) triples is the format of this external storage.

Given this external format, the layer-centered scheme provides superior performance when the system loads layers from secondary storage into main store. With the layer-centered scheme, the external dictionary can simply be treated as a remote disk page that is mapped into main memory. But if the node-centered scheme is employed, the values in the layer dictionary must be asserted in each node. The result is that loading process must sweep large portions of the virtual memory to access the specified nodes.

Another virtue of the layer-centered scheme relates to the grouping of information. Layers organize coordinated values. Hence, a system designer can specify that attributes from different objects must be swapped into main memory at the same time by placing them in the same layer. This is useful when subsets of the attributes of an object are used for some purposes and not for others. The present implementation of Smalltalk treats an object as an indivisible unit, swapping the entire object at one time. The layer-centered design allows a finer granularity in memory management. The attribute, not the object, is treated as the smallest memory element. The result is that objects are no longer integral blocks of storage. Consequently, the traditional discussion of a Smalltalk system as an interacting collection of integral objects is no longer appropriate. In PIE, a system is an interacting collection of partial descriptions of objects.

6. Conclusions

The existence of a software problem is widely acknowledged: it is becoming progressively more costly to develop and maintain software systems. Their complexity is growing faster than our ability to manage these systems. We have argued that the use of layered networks to represent design alternatives is one means for managing this complexity for three major reasons: (1) Layered networks are a more flexible tool to represent an evolving design than present file systems. (2) The cognitive complexity for employing these networks can be managed by suitable user interfaces. (3) The computational cost in storage space and retrieval time is reasonable.

Traditionally, structured programming and high level programming languages are offered as means to improve the software development process. Both are based on explicitly representing the structure of a design—structured programming emphasizes modular decomposition while high level programming languages emphasize the description of a module in terms of vocabulary closer to its intent. These techniques are important. However, neither addresses the orthogonal issue of coping with the evolution of a design. Neither is concerned with coordinating change across modules nor do they address the need of a designer to examine alternatives.

Advanced programming environments have tools for describing a system's current state—indexing programs for cross references, file packages for automating the storage and retrieval of a system—but again they are deficient in tools for describing the process of change.

Software evolves. It goes through a life cycle of design, implementation, and redesign. Layered networks are a means to represent an evolving structure and are therefore a useful basis for a software environment. Elements of a software system are represented by nodes in the network: structure by labelled links between these nodes. Modifications to the structure are represented by layers. In particular, refinements are captured by the domination of layers within a context while alternatives are represented by contexts containing different layers. Shared structured is expressed by shared layers.

The techniques described in this paper have been tested in the context of the Smalltalk programming environment. However, they can be applied to any programming system that provides a layered network database and an interactive display interface. We believe that the payoff of such databases for software design and development will more than justify their cost in terms of retrieval time and storage.

7. Bibliography

[Bobrow & Winograd, 1977]

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* (August 1977), 213-222.

[Birtwistle, 1973]

Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, C., *Simula Begin*, Auerbach, Philadelphia, 1973.

[Cohen, 1975]

Cohen, P., "Semantic Networks and the Generation of Context", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Tsibilisi: (1975), 134-142.

[Glasser, 1978]

Glasser, Alan L., "The Evolution of a Source Code Control System", in S. Jackson and J. Lockett (eds.), Proceedings of the Software Quality and Assurance Workshop, ACM, (1978), 122-125.

[Goldstein & Roberts, 1977]

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.

[Hendrix, 1975]

Hendrix, Gary G., "Expanding the Utility of Semantic Networks Through Partitioning", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Tsibilisi: (1975), 115-121.

[Hewitt, 1971]

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).

[Ingalls, 1978]

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), 9-16.

[Ivie, 1977]

Ivie, E.L. "The Programmers Workbench—A Machine for Software Development." *Communications of the ACM*, V. 20 No.10 (October 1977), 746-53.

[Kay, 1974]

Kay, A., SMALLTALK, "A Communication Medium for Children of All Ages". Palo Alto, California: Xerox Palo Alto Research Center, Learning Research Group (1974).

[McDermott, 1974]

McDermott, D.V., "Assimilation of New Information by a Natural Language-Understanding System". AI-TR-291 MIT-AI Laboratory (February 1974).

[Rochkind, 1975]

Rochkind, Marc J., "The Source Code Control System", *IEEE Transactions on Software Engineering* (December 1975), 364-370.

[Rulifson, 1971]

Rulifson, J., Waldinger, R., and Derksen, J., "A Language for Writing Problem-Solving Programs", IFIP, 1971.

[Sussman, 1972]

Sussman, G., & McDermott, D., "From PLANNER to CONNIVER—A Genetic Approach". *Fall Joint Computer Conference*. Montvale, N. J.: AFIPS Press (1972).