Designing Communities of Data Types

Wilf R. LaLonde

SCS-TR-73
May 1985

School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

This research was supported by NSERC (Natural Sciences Engineering Research Council) and DREA (Defense Research Establishment at Atlantic)

Designing Communities of Data Types

Wilf R. LaLonde

School of Computer Science

Carleton University

Ottawa, Ontario, Canada K1S 5B6

Abstract

With the increasing availability of object-oriented languages and processors, a small but growing community of users is slowly gaining experience with the object-oriented paradigm. With the realization that all programming is concerned with designing and implementing data types, there is a growing awareness that data type designing is far more difficult than traditional books on data stuctures and data types have suggested.

Designing data types in isolation is fundamentally different from designing them for integration into communities of data types. New approaches and methodologies are needed for handling the design of communities and sub-communities of data types. A modest beginning at uncovering and highlighting some of the issues is presented under the guise of designing a community of List data types.

1 Introduction

> Smalltalk exemplifies the object-oriented movement and the message passing paradigm.

The object-oriented paradigm is becoming increasingly popular. This is due partly to the recent availability of textbooks and processors associated with Smalltalk[1,2,3], partly to the ongoing research into object-oriented languages[4,5,6,7,8], partly to the novel applications for the paradigm[9,10,11,12], and partly to the increased emergence of object-oriented facilities in other well-known languages: the Flavour system in Lisp[14,15], extensions of C^{OopC}, and Prolog[16,17].

This small but growing community of users is slowly gaining experience with object-oriented software. With the realization that all programming in the paradigm is concerned with designing and implementing data types, there is a growing awareness that data type designing is far more difficult than traditional books on data stuctures and data types have suggested. This is most apparent in users with experience beyond that of "data types in the small" – implementers of the Smalltalk system software or the Symbolics Lisp machine software, for instance.

As data type designing shifts from the narrower concerns of "data types in the small" – the emphasis and focus of more traditional languages – to "data types in the large"[18], it is quite clear that the "art" is far from understood and that existing programming languages are far from providing adequate support.

To uncover some of the important issues that must be addressed when designing interrelated communities of data types, we focus on designing Lists, a data type well-known to the Artificial Intelligence community. The ideas could equally well have been promoted by attempting to design a community of arrays or even stacks. We emphasize that it is not a detailed design that we are focusing on but on the detailed issues that must or should be addressed when attempting to construct such a design. The development assumes a cursory knowledge of Smalltalk – the basic philosophy of the language and the notion of inheritance. It does not require knowledge of the syntax of Smalltalk.

2 Designing the List Operations

> A preliminary specification that ignores the detailed semantics is a reasonable first step to designing lists.

The object-oriented viewpoint treats objects as the primary focus in designing and implementing software systems. Objects are endowed with a representation (data) and methods (code) for responding to messages (queries or operations). Accepted wisdom dictates that we design the messages first (i.e., the specification) with the implementation (the methods) and representation last. Because it is exploratory in nature, the initial specification should gloss over

intimate semantic details. These are not to be forgotten but rather left for a later more important step. A reasonable first pass might provide the following set of operations:

Empty? (AList) ⇒ Boolean: Is the list empty?
Length (AList) ⇒ Integer: The number of elements in the list.

== (AList, AnObject) ⇒ Boolean: True iff the object is identical to (the same object as) the list.
= (AList, AnObject) ⇒ Boolean: True iff the object is equal to the list.

First (AList) ⇒ Object: The first element of the list.
Last (AList) ⇒ Object: The last element of the list.
Element (AList, Position) ⇒ Object: The element of the list at the specified position.

Rest (AList) ⇒ List: The remaining (all but first) elements of the list.
AllButLast (AList) ⇒ List: The remaining (all but last) elements of the list.
Prefix (AList, NewSize) ⇒ List: An initial portion of the list.
Suffix (AList, NewSize) ⇒ List: A final portion of the list.
Sublist (AList, Start, NewSize) ⇒ List: A subportion of the list.

Precede (AList, AnObject) ⇒ List: A new list with AnObject as the first element.
Follow (AList, AnObject) ⇒ List: A new list with AnObject as the last element.
Append (AList, AnotherList) ⇒ List: A new list obtained by appending the two.

Write (AList, AFile) ⇒ List: The list is output and returned.

Do (AList, ABlock) ⇒ Nothing: Invoke the one-parameter block (function) on each list element.
DoWhen (AList, ABlock, WhenBlock) ⇒ Nothing: Invoke the one-parameter block (function) on each list element for which the when-block returns True.
Collect (AList, ABlock) ⇒ List: Invoke the one-parameter block (function) on each list element and collect the answers into a list.

Table 1: A Cursory Design of a Set of List Operations

In an object-oriented setting, function invocation is really a message being sent to a distinguished object. If the language uses a traditional syntax, this would be a distinguished parameter – typically the first parameter. Thus "Append (A, B)" is interpreted as an append message to object A with B as the auxiliary information. In the case of Append, both parameters are lists and the ordering of the parameters is natural.

On the other hand, consider an operation which takes an arbitrary object O and a list L and constructs a new list in which the first element is O and the remaining elements are the same as those in L. In Lisp, this operation is Construct[1] and would be invoked as "Construct (O, L)". According to the object-oriented view, this is a message destined for O; i.e., it is interpreted as an object operation rather than the more appropriate list operation. For it to be a list operation, the

[1] Actually, the operation is called CONS.

first parameter must be a list. Unfortunately, we can't just write "Construct (L, O)" because this has the undesirable connotation that the object is added at the right end instead of the left (the front of the list) as intended. We must choose a more appropriate name; "Precede (L, O)" is suitable. It also highlights the need for the complementary operation invoked as "Follow (L, O)". *Name design is an important consideration at this early stage.*

The above is a reasonable start though not a complete set of operations. For instance, the ~== (not of ==) and ~= (not of =) operations are omitted; so are operations for interrogating the type of an instance, for making a copy, for converting to other types. Additional enumeration operations beyond Do, DoWhen, and Collect could also be provided.

3 Partitioning The Operations In A Hierarchy

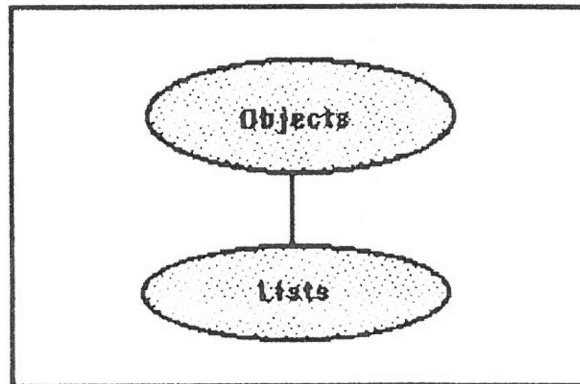
Specialization and generalization are fundamental organizational approaches for designing in the large.

The first step to designing in the large is the observation that some of the operations are more general than list operations. The design space for operations can be split into two: those operations applicable to all objects and those applicable only to lists.

In particular, the comparison operations and the write operation appear to be applicable to more than just lists. Additionally, many of the operations omitted can be seen to be Object operations. *One aspect of designing in the large is the proper partitioning of operations between classes of objects.* In this case, lists are a specialization of objects; i.e., all operations applicable to objects are also applicable to lists but lists provides additional operations that are unique. An inheritance mechanism such as provided in Smalltalk is ideally suited for this level of detail.

4 Attention To Semantic Details

Attention to the detailed semantics of the operations might seem like mere perseverance



A Simple Inheritance Network

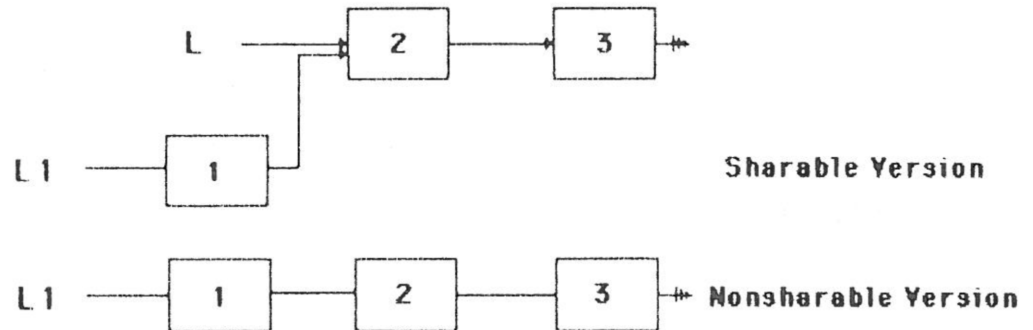
Figure 1

but it can open up a whole new pandora's
box of design problems.

In the previous section, we ignored the detailed semantics of the operations. The stage is now set for a more careful analysis. A reasonable set of operations to consider as a unit is the triple consisting of First, Rest, and Precede. If L is the list containing elements 2 and 3, and L_1 is obtained by computing "Precede (L, 1)", two lines of inquiry are possible: one concerned with whether or not the operations are destructive and the other concerned with sharability.

One possible design decision is to stipulate that none of the specified operations be destructive. Thus the above Precede operation would not have any side effects on L. Destructive operations could still be introduced but more appropriate names should be provided; e.g. DestructivePrecede.

With respect to sharability, two possibilities arise: list L_1 could be constructed so as to contain 1 and "share" the contents of L or it could be an entirely new list with no sublist in common with L as shown in Figure 2. This is not an implementation issue because it has to do with the detailed semantics of the operations.



Sharable and Nonsharable Lists

Figure 2

With the sharable variety of lists, the Precede operation must have non-copying semantics; i.e., it constructs a new list object whose primary role is to encapsulate its two components: the first element and the remaining elements. Consequently, the First operation is able to return the first element as it existed originally (not a copy) and the Rest operation returns the list originally provided to Precede. This is equivalent to saying that

$$\text{First (Precede (L, O))} == \text{O}$$
$$\text{Rest (Precede (L, O))} == \text{L}$$

Sharable lists are consequently extremely space and time efficient – one of the primary reasons for its popularity in Lisp. But this isn't the final word on sharability. In particular, consider the complementary operation Follow. If we construct lists L_1 and L_2 so that they share the same list L, can "Follow (L_1, O)" share any portion of L_1? If the lists are intended to share as in Figure 2, the answer is no unless destructive operations are allowed. We could for instance destructively add O to the right end of list L_1. Unfortunately, the side effect is pervasive; i.e., both L_1, L_2, and L have been modified. By insisting the operation have no side effects (a previous design choice), we have no recourse but to copy L_1; i.e., to construct a new list containing all of

the elements of L_1 including O.

It is quite clear that the sharable lists are actually suffix-sharing lists. Such lists can easily be extended on the left (using Precede) but not on the right (Follow is expensive). Moreover, the first element (using First) is easily obtained but the last element (using Last) can only be reached by sequencing through all preceding elements. Similarly, Rest is an inexpensive operation but AllButLast must construct a new list with the required elements.

However, because of the symmetrical nature of these operations, it should be obvious that a prefix-sharing variety of lists can be easily devised. Such lists could be constructed easily by adding to the right and easily traversed from right to left (using Last and AllButLast); First, Rest, and Precede in their case would be expensive. Figure 3 illustrates these two kinds of sharing.

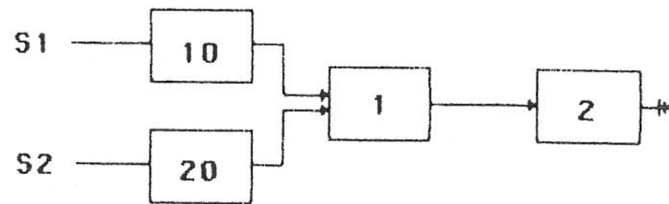
If we now consider the nonsharable variety, operations like Precede and Follow must be expensive since new lists must be constructed – no possibility of sharing exists. Such lists would likely be constructed with destructive operations instead or as the result of a conversion from the sharable variety. On the other hand, accessing order (left to right or right to left) could be equally inexpensive (using Element however – not using Rest or AllButLast). Of course, a more complex representation would have to be devised to support these operations.

5 Multiple Views: Classes Versus Varieties

> In a large community of Smalltalk users, the name space becomes bewilderingly large.

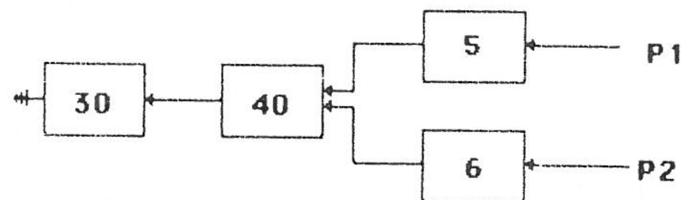
The three varieties of lists discussed above is not a complete inventory of possible designs. Special applications are likely to come up with additional varieties having very specific space-time tradoffs. Using the Smalltalk notions of specialization, we obtain a more elaborate class hierarchy as shown in Figure 4. In the Smalltalk tradition, lists become an abstract class; i.e., a class for which no instances can be created. Only instances of Object, PrefixSharingList, SuffixSharingList, and NonSharableList are possible.

When considered in isolation, it is easy for a user to be or to become knowledgeable about a specific data type and its many specializations. However, there may be little incentive to do so



S1 = #SuffixSharingList (10 1 2)
S2 = #SuffixSharingList (20 1 2)

S1 and S2 share the sublist containing 1 and 2



P1 = #PrefixSharingList (30 40 5)
P2 = #PrefixSharingList (30 40 6)

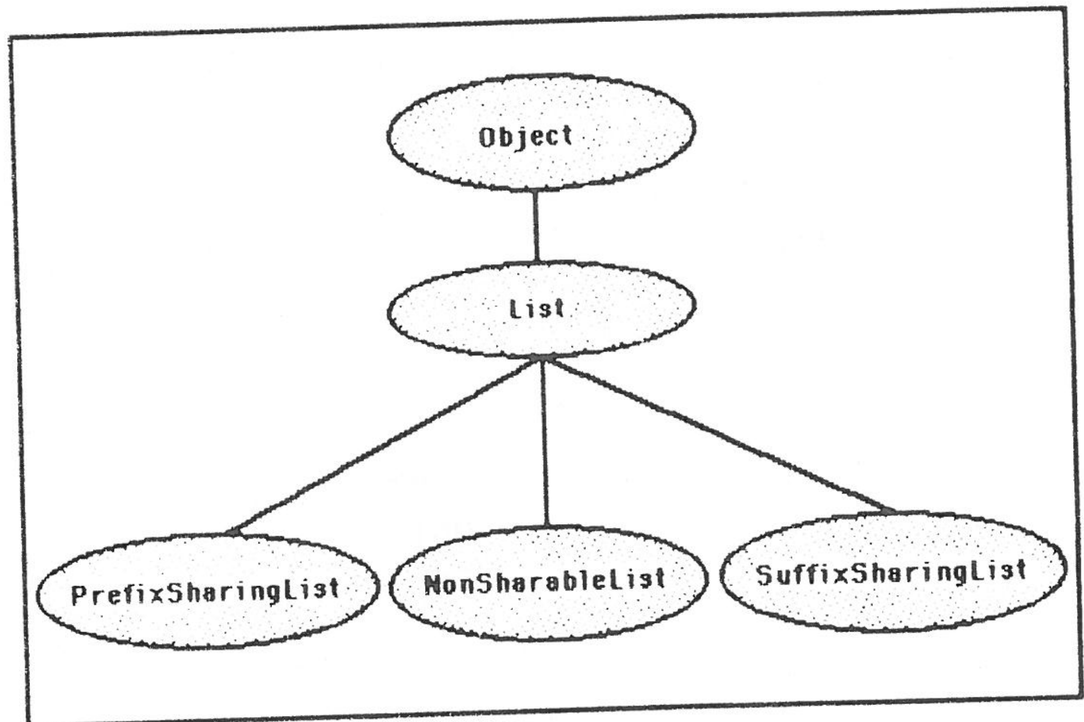
P1 and P2 share the sublist containing 30 and 40

Prefix-Sharing and Suffix-Sharing Lists

Figure 3

given that a standard specialization is generally used. Other specializations may only be considered when specific space/time tradeoffs need to be met.

In the context of a large evolving system, understanding the system means being able to browse through the class hierarchy. Even with a sophisticated tool, the hierarchy can be bewildering in the presence of a large number of classes. For a sampling, consider lists with 3 or 4 specializations, arrays with 5 or 6 specializations, stacks with 2 or 3 specializations, sets with 2 or 3 specializations, windows with ... *A small name space is essential if a user is to be able to*



A List Inheritance Network

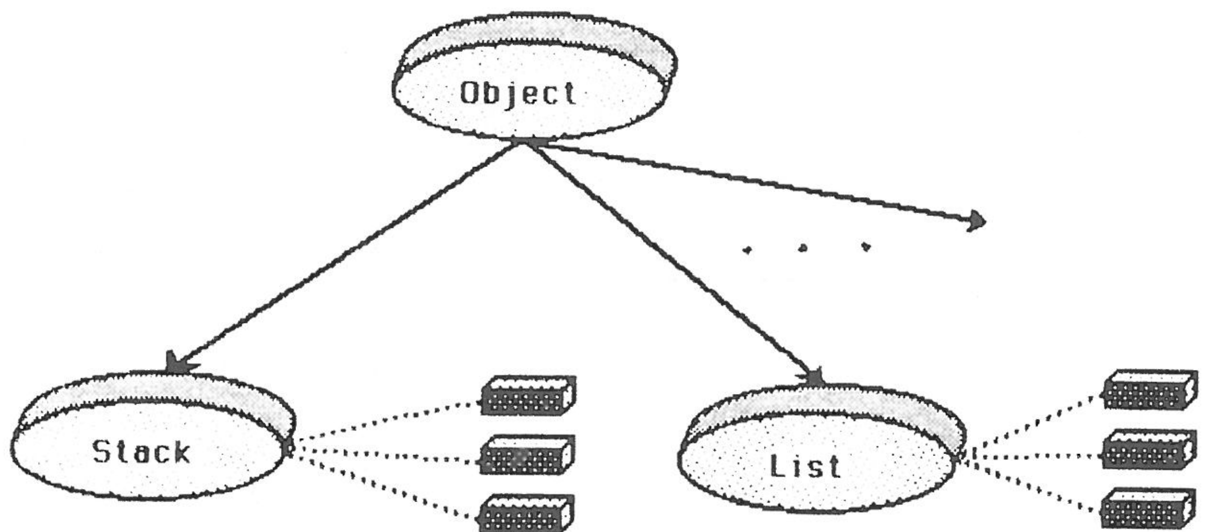
Figure 4

understand the entire system.

To begin to address this problem, we need to distinguish between generic classes and their specializations – call these **varieties** to emphasize their special role. In the more traditional view, classes and varieties are totally intermixed, each playing interchangeable roles. A variety is itself viewed as a class in the context of a subclass. The distinction is a convenience enabling discussions to be better focused. However, an important distinction can be made if we acknowledge that the two views play different roles.

The **class view** describes the relationship between individual classes in the generic sense whereas the **variety view** describes the specific special cases of a generic class. More important, the variety view does not contribute to the name space of the class view. Alternatively, the class view contributes to a **global** name space whereas the variety view contributes only to a **local** name space.

By way of illustration, consider Figures 5 and 6. The class view of Figure 5 (when suitably enlarged to include all useful data types) still provides a small manageable name space. Correspondingly, details of a specific data type (Figure 6) are provided in the variety view.

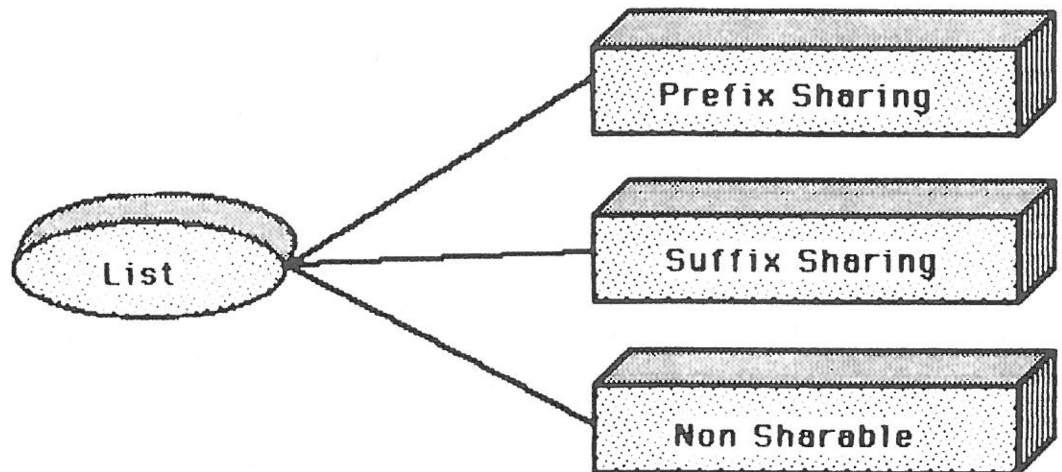


The Class View

Figure 5

6 When Should Varieties Be Migrated to Classes?

When a non-standard variety gains



The Variety View

Figure 6

prominance, it may be time to promote it to
a class of its own.

If we continue our design of lists by considering the issue of sharability in more detail, additional issues begin to surface. For instance, consider operation Prefix in the context of suffix-sharing lists. The traditional Lisp semantics dictate that taking a proper prefix requires a copy; i.e., suffix-sharing lists cannot share prefixes. The reason for this stance is simple: by not allowing prefixes that share, there is no need to keep track of size information. Intuitively, lists continue until they end.

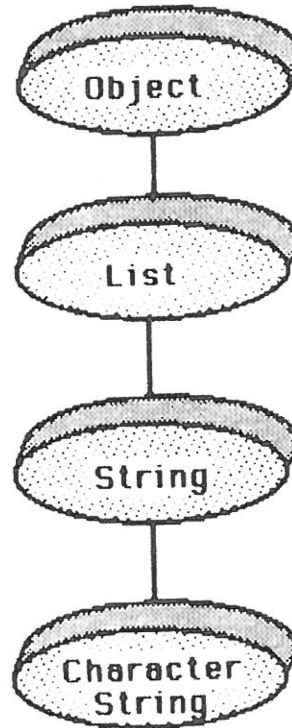
If the Prefix operation were allowed to return a list that shares the original list, it would be necessary to keep track of the size. Is there a precedent for a variety of lists with this property? The answer is yes. Strings when implemented for maximum sharability have this property. In particular, the variety of strings implemented in Xpl[19] have exactly this property. Of course, the

traditional notion of strings considers only elements that are characters. Allowing arbitrary elements is an obvious way to generalize.

But is it appropriate to use the term string to refer to suffix-sharing lists with prefix-sharing capabilities? We need to investigate typical string operations to resolve the issue. A quick glance at Table 1 reveals that the operations indeed are typical of strings. The only anomaly is that operation Sublist ought to be called Substring in the string context. It therefore seems reasonable to consider suffix-sharing strings as a variety of lists. Note that suffix-sharing strings are not equivalent to prefix-sharing strings; i.e., suffix-sharing lists with prefix-sharing capabilities are not equivalent to prefix-sharing lists with suffix-sharing capabilities. The Follow operation is still expensive for the former while it is the Precede operation which is expensive for the latter. Our design so far includes the following:

- prefix-sharing lists (no suffix-sharing capability)
- prefix-sharing strings (with suffix-sharing capability)
- suffix-sharing lists (no prefix-sharing capability)
- suffix-sharing strings (with prefix-sharing capability)
- non-sharable lists

We could continue in this manner adding two varieties of character strings to the design. However, the importance of strings suggests that they be promoted to a more visible role. In particular, both general strings and character strings should be evident in the global name space. Compared to lists, strings have additional requirements (they provide more sharing capability). It makes sense to view strings as a specialization of lists. Similarly, character strings are a specialization of strings since they have the additional requirement that the elements all be characters. The result is a class view that includes the classes of Figure 7 where each class (lists, strings, and character strings) contains the three varieties of Figure 6. We might now observe and question whether or not there is any difference between non-sharable lists and non-sharable strings. In our opinion, they are identical.



The List/String Hierarchy

Figure 7

7 The Role of Classes In The Design

> A class is a shepherd watching over many varieties of sheep.

Classes in traditional programming languages do not enjoy the same freedoms as instances; e.g., few (if any) operations are defined on classes. Typically, compilers are able to compare variable types for equality but user code is not provided with the capability. In object-oriented languages, classes can be manipulated as easily as traditional instances. This is even more apparent when we consider that classes themselves are instances of some other class – generally termed a metaclass. Classes, like their instances, are provided with useful operations. Typically, these operations have to do with the creation, maintainance, and destruction (usually automatic in garbage collected systems) of instances of the class. They also provide answers to useful queries about the class.

It is the Stack class that responds to a message for a new stack, the Set class that provides new sets, the Array class that provides new arrays, ... Each class can enumerate (or refuse to enumerate) its instances, provide statistics on the typical size of an instance, the number of instances, ... They can also respond to class specific queries. SmallIntegers, for example, can provide the maximum or minimum small integer.

By partitioning the system into classes and varieties, we make it clear that the classes provide the mechanism for selecting appropriate varieties. The language of selection need not correspond one-to-one with the different varieties of the class. For instance, we could envision obtaining an empty list (the seed from which longer lists can be constructed) in any one of the following ways:

- **List new**
- **List new:** #standard
- **List new:** #suffix-sharing
- **List new:** #left-to-right-traversable
- **List new:** #right-to-left-constructable
- **List new:** #prefix-sharing
- **List new:** #right-to-left-traversable
- **List new:** #left-to-right-constructable
- **List new:** #non-sharable
- **List new:** #bi-traversable
- **List new:** #bi-constructable
- **List new:** #non-sharable **size:** 20

The first four choices all select suffix-sharing lists, the next three – prefix-sharing lists, and the last four – non-sharable lists. The last choice provides an initial estimate of the maximum list size; in general, more than one variety of sharable lists could be provided for different requirements.

With complicated data types, the proper design of a selection language or notation can be an issue on its own. Designing a selection language that is less revealing of the varieties is a worthwhile goal. Better use of terms and combinations like #fast, #small, #(fast, bounded), #(fast, unbounded), ... might prove useful for types such as arrays.

8 The User Versus The Implementer

> The user should interact with the classes –
> the implementer, with the varieties.

The traditional focus for designing data types in the small is the operations. The user is provided with a complete list of operations with detailed (although typically informal) semantics. The implementer, on the other hand, chooses a suitable representation and realizes the operation with actual code. The design that results is called an **abstract data type** to distinguish it from its less desirable alternative – the **concrete data type** in which the representation is divulged. Changing the representation when it is known to users is very difficult to achieve without negotiation.

In a larger setting, the same view still prevails but the increased complexity demands additional facilities. In particular, it is not sufficient for a user to be provided with a complete description of the operations for prefix-sharing lists and a corresponding description for suffix-sharing lists in isolation. He needs to have information about lists in general and also information about differences between the varieties. An online help system is needed to provide detailed explanations and examples. Advantages, disadvantages, and peculiarities of the different varieties must be explained.

A natural repository for this information is the class since it is a focal point for the different varieties of the type. Not only does the user interact with the class for obtaining instances of the

class but also for obtaining information about the instances. While users interact primarily with the class, implementers interact primarily with the varieties.

9 Organizing Hierachies: Distinguishing Class Hierarchies from Operation H

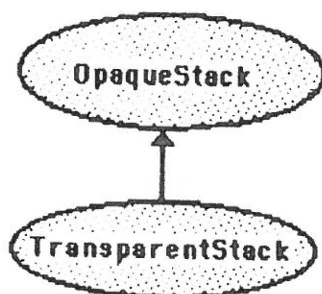
The class and operation inheritance hierarchies can cause confusion especially when they have different inheritance directions.

If the number of different varieties and their relationship is sufficiently complex, a hierarchy of varieties can be designed. At the implementation level, the focus is no longer on the class but on the representation and realization of the operations. In particular, it is worthwhile evaluating preliminary proposals for the representations of the different varieties and the corresponding operations to relate both their differences and their commonalities. For some data types, the result of the evaluation is sometimes surprising. This can be best illustrated in the context of stacks.

Suppose our stack design consisted of two unbounded varieties: (1) opaque stacks with typical operations such as Push, Pop, Empty? and (2) transparent stacks with the additional operation Peek. Additional varieties would likely be provided but two are sufficient to illustrate the interaction. With respect to the class relationship, it should be clear that transparent stacks are a special case of opaque stacks; it has the same operations as opaque stacks and more. The associated class hierarchy is therefore as depicted in Figure 8.

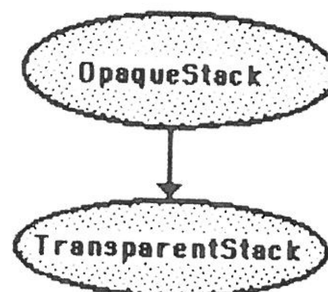
From an implementation standpoint, a reasonable goal is to minimize the amount of implementation code. This can be achieved by choosing one of these data types as the implementation base and then implementing the other in terms of the first. Since they differ in only one operation, we can either (1) implement opaque stacks as the base, have transparent stacks inherit both the representation and the operations from opaque stacks, and add the additional operation Peek or (2) implement transparent stacks as the base, have opaque stacks inherit from transparent stacks, and remove the disallowed operation.

The former approach violates the principles of abstract data types since the Peek operation



A Stack Class Hierarchy

Figure 8



A Stack Operation Hierarchy

Figure 9

cannot be implemented without detailed knowledge of the representation. On the other hand, the representation need not be known to implement the latter strategy; depending on the mechanisms available, the Peek operation could be (1) removed explicitly, (2) removed by making it private, or (3) removed by re-implementing it to signal an error. Using case (3), for instance, opaque stacks would inherit all operations from transparent stacks as shown in Figure 9. It would provide its own error-reporting version of Peek. No representation inheritance is needed since details of the representation are not required.

In a language like Smalltalk, there is only one inheritance mechanism which combines class inheritance, representation inheritance, and operation inheritance. Consequently, implementers are forced to violate either the principles of abstract data types or the definition of class specialization by making opaque stacks subordinate to transparent stacks. The latter risks confusion on the part of the user. Most designers would opt for the former under the circumstances. Of course, these distinctions are purely academic if the user is not privy to the design structures of the implementer. In particular, the class view can describe the relationships between the different varieties using class inheritance. The variety view can totally violate these relationships as long as the user's class view is maintained.

10 Partitioning The Operations: Primitives Versus Non-Primitives

> Representation accessing primitives embody the core semantics of the operations – non-primitives round out this basic set of operations.

Under the assumption that a design is always subject to further refinements and extensions and consequently never complete, it is useful to attempt a partition of the operations according to the criterion "Does it need to know the representation?". Those that require direct access to the representation, we will call **primitives** and the others, **non-primitives**. Future modifications to the representation should impact only the primitives.

In general, the dichotomy between primitives and non-primitives is not clear-cut. Drastically changing the representation can cause major reorganizations to an existing partitioning scheme. Additionally, even though an operation can be implemented non-primitively, there may be compelling efficiency considerations dictating otherwise.

If we consider suffix-sharing lists, for instance, it seems reasonably clear that the representation must include a FirstComponent, a RestComponent, and an Empty?Component. The latter component, for example, is essential if many instances of empty lists are to be provided. On the other hand, it could be avoided by insisting that only one empty list be allowed; this is the traditional Lisp solution. With this representation, it seems reasonably clear that Empty?, First, Rest, and Precede must be primitive. Verifying that the others are non-primitive amounts to sketching their implementation in terms of the primitives (or other non-primitives). For instance, operations Last, Follow, Prefix, and Suffix could be implemented as follows:

```
operation
   Last (AList)
begin
   select
      choice Empty? (AList)
         error "operation illegal on empty list"
      choice Empty? (Rest (AList))
         return First (AList)
      choice otherwise
         return Last (Rest (AList))
   endselect
```

```
endoperation

operation
   Follow (AList, AnElement)
begin
   select
      choice Empty? (AList)
         return Precede (Empty (List, #suffix-sharing), AnElement)
      choice otherwise
         return Precede (Follow (Rest (AList), AnElement), First (AList))
   endselect
endoperation

operation
   Prefix (AList, NewSize)
begin
   select
      choice NewSize < 0 or (NewSize > 0 and Empty? (AList))
         error "illegal size"
      choice NewSize = 0 or Empty? (AList)
         return Empty (List, #suffix-sharing)
      choice otherwise
         return Precede (Prefix (Rest (AList), NewSize - 1), First (Alist))
   endselect
endoperation

operation
   Suffix (AList, NewSize)
begin
   let
      OldSize be Length (AList)
   in
      select
         choice NewSize < 0 or NewSize > OldSize
            error "illegal size"
         choice otherwise
            for I <- NewSize + 1 to OldSize do
               AList <- Rest (AList)
            endfor
            return AList
      endselect
   endlet
endoperation
```

Because of its mirroring nature, a similar partitioning for prefix-sharing lists contains Empty?, Last, AllButLast, and Follow as primitives and all others as non-primitives. Unfortunately, there is little opportunity to share operations. For example, even though both provide Prefix as a non-primitive, neither implementation is sufficiently general to be useable by the other. Prefix for suffix-sharing lists is implemented in terms of Empty?, First, Rest, and

Precede whereas for prefix-sharing lists, it is implemented in terms of Length and AllButLast.[2] Partitioning operations into primitives and non-primitives did not simplify the implementation in this case but it did serve to highlight an important distinction. Adding new non-primitives should be straigthforward and matter of fact. Adding a new primitive should be cause for concern. In particular, new primitives typically signal changes and/or extensions to the core semantics of the variety. Greater care is required to ensure that the side effects of the changes are intentional.

As the number of varieties of a class increases, the potential for sharing non-primitive operations among several alternatives also increases. Sometimes this sharing can be restricted to the varieties within a class; at other time, the sharing is more distributed.

11 Sharing Non-Primitives: A More Robust Implementation

> Mechanisms for sharing is both the source of power and the source of all problems for programming in the large.

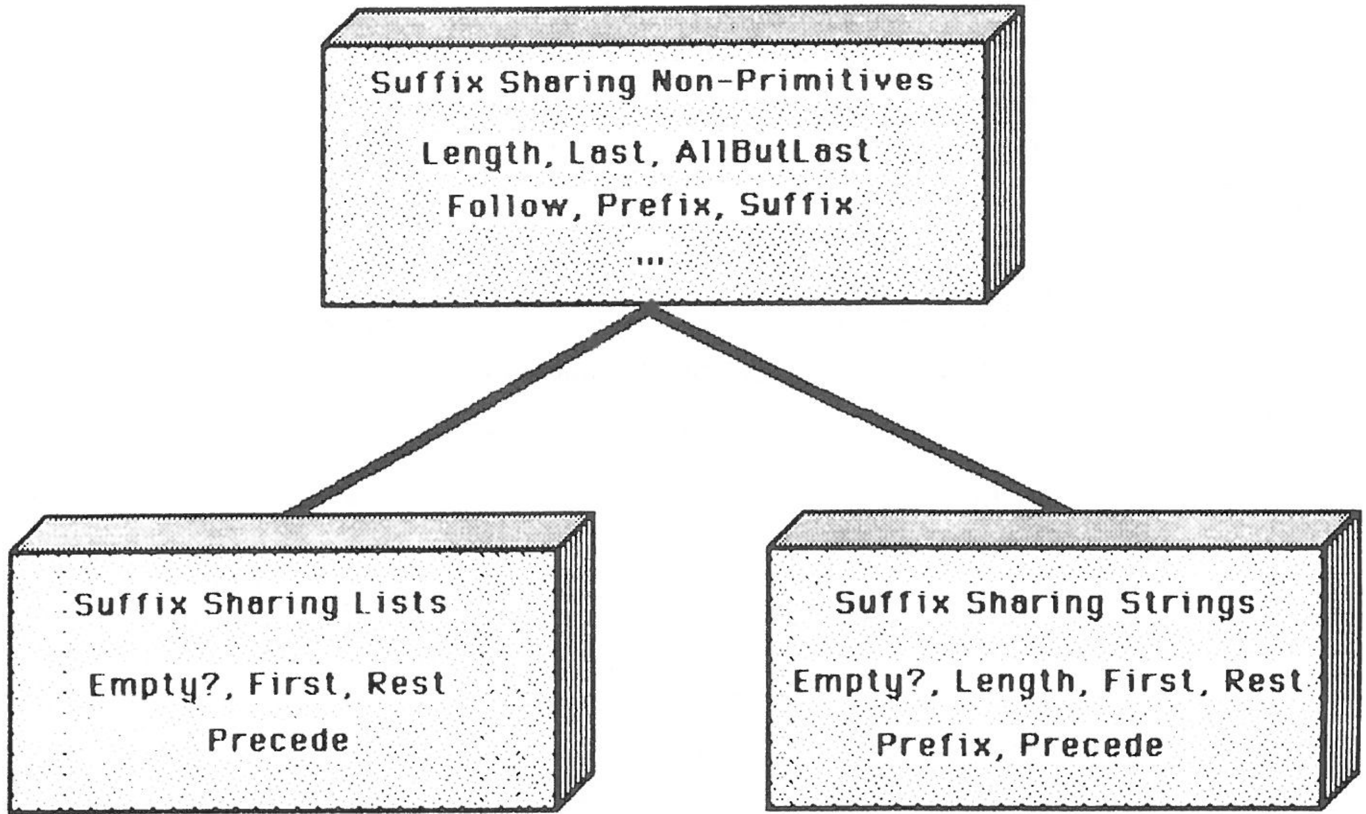
Because suffix-sharing strings were designed as a special case of suffix-sharing lists,[3] it's no surprise that primitives and non-primitives can be partitioned along similar lines. The partitioning, however, it not identical. In particular, Prefix for strings must be primitive since it must have access to that part of the representation concerned with encoding the sharable portion of the string. Typically, this is a length field although a pointer to the end of the string could also be used. Additionally, with the former encoding, there is no point in computing the length of a string using Empty? as is the case for lists. On final analysis, suffix-sharing strings have primitives Empty?, Length, First, Rest, Prefix, and Precede; when pressed, Empty? might be also viewed as a non-primitive.

All other operations, like Last, Follow, and Suffix shown above, apply equally to lists and strings. It seems worthwhile packaging these operations so they can be shared by both lists and strings. If the list version of Length and Prefix is also provided as part of the package, lists and strings can be implemented by providing only the list and string primitives respectively. In the

[2] Although not shown, it is the dual of the above Suffix operation.

[3] The same arguments apply to prefix-sharing strings and lists.

Smalltalk terminology, we are defining an abstract class containing operations applicable to both lists and strings. Lists provides its own operations and inherits from the abstract class; strings do the same but in addition override the Length and Prefix operations provided by the abstract class. The result is shown in Figure 10.



Abstract Classes For Sharing Suffix-Sharing Non-Primitives

Figure 10

12 Prototypes Versus Classes

> In a sophisticated environment where objects can migrate, reorganize, self-optimize and be transformed, the simplistic class notion is inadequate.

In general, a class mechanism enforces the restriction that all objects of a specific class have the same representation. This is a rather strong requirement when we are only interested in the behaviour of a group of objects. The alternative is to base behaviour on the notion of **prototypes** or **exemplars**; i.e., special sample objects defined specifically to serve as a model for other objects. When new objects playing the same role as exemplars are needed, clones are generated for use.

In this paradigm, each variety of objects is governed by a collection of exemplars. The traditional approach is obtained by using only one exemplar. In the presence of large objects, for example, it is convenient to have two representations: a traditional in-core representation and a moved-to-disk representation. We could also contemplate designing access monitoring objects that decide on an appropriate representation dynamically.

For more traditional applications, exemplars allow run-time selection of special case code to be factored out and the run-time test to be removed entirely. To continue with our example, consider suffix-sharing lists in more detail. We previously came to the conclusion that such lists had to have at least three components to the representation if multiple empty lists were to be allowed: an empty? component, a first component, and a rest component. Additionally, by not allowing multiple empty lists, we could remove the need for the empty? component. Using prototypes, we instead define two exemplars: an EmptySuffixSharingList exemplar and a NonEmptySuffixSharingList exemplar. The corresponding primitives can be implemented as follows:

```
exemplar
   EmptySuffixSharingList
begin
   representation
   endrepresentation

   operation
      Empty? (AnEmptyList)
```

```
            begin
               return true
            endoperation

         operation
            First (AnEmptyList)
            begin
               error "operation not allowed on empty lists"
            endoperation

         operation
            Rest (AnEmptyList)
            begin
               error "operation not allowed on empty lists"
            endoperation

         operation
            Precede (AnEmptyList, AnElement)
            begin
               return Initialize (Clone (NonEmptySuffixSharingList), AnElement, AnEmp-
tySuffixSharingList)
            endoperation
      endexemplar


      exemplar
         NonEmptySuffixSharingList
      begin
         representation
            FirstPart
            RestPart
         endrepresentation

         operation
            Empty? (ANonEmptyList)
            begin
               return false
            endoperation

         operation
            First (ANonEmptyList)
            begin
               return FirstPart
            endoperation

         operation
            Rest (ANonEmptyList)
            begin
               return RestPart
            endoperation

         operation
            Precede (ANonEmptyList, AnElement)
            begin
               return Initialize (Clone (NonEmptySuffixSharingList), AnElement, ANonEmp-
tySuffixSharingList)
            endoperation

         operation private to AnEmptyList, ANonEmptyList
            Initialize (ANonEmptyList, AnElement, AnotherList)
            begin
```

```
            FirstPart <- AnElement
            RestPart <- AnotherList
            return ANonEmptyList
        endoperation
endexemplar
```

Note: only the receiver of the message (the first parameter) has access to its representation. As in Smalltalk, a special Initialize operation is consequently required to set the components of the representation. In an object-oriented language, looking up the Empty? operation, for example, does not depend on whether the receiver is an empty list or not. Once, the specific receiver is determined, however, the answer is immediate. Run-time testing for the empty list is removed.

The same idea can also be applied to the non-primitives. For instance, the Prefix operation could be partitioned into two: one for empty lists and another for non-empty lists. The process is reminiscent of the partitioning process that Prolog programmers must deal with.

```
nonprimitive
    EmptyPrefixSharingList, EmptySuffixSharingList
begin
    operation
        Prefix (AnEmptyList, NewSize)
    begin
        select
            choice NewSize ~= 0
                error "illegal size"
            choice otherwise
                return Clone (AnEmptyList)
        endselect
    endoperation
endnonprimitive


nonprimitive
    NonEmptyPrefixSharingList, NonEmptySuffixSharingList
    NonEmptyListNonPrimitives
begin
    operation
        Prefix (ANonEmptyList, NewSize)
    begin
        select
            choice NewSize < 0
                error "illegal size"
            choice NewSize = 0
                return EmptiedClone (ANonEmptyList)
            choice otherwise
                return Precede (Prefix (Rest (ANonEmptyList), NewSize-1), First (ANonEmptyl

        endselect
    endoperation
endnonprimitive
```

Designing nonprimitives to apply to several data types is an evolutionary process. For instance, the initial version of Prefix was designed only for use with suffix sharing lists. Consequently, a new empty list was constructed via "Empty (List, #suffix-sharing)". Generalizing Prefix so that it would also apply to prefix sharing lists required a modification. In the empty list case, direct reference to the specific variety of lists could be avoided by cloning the first parameter. If not already available, the clone operation was sufficiently important to provide it at the highest possible level – making it an object operation. For the non-empty list case, the situation was similiar but no existing empty list was available for cloning. Consequently, a new primitive was invented that provided the capability.

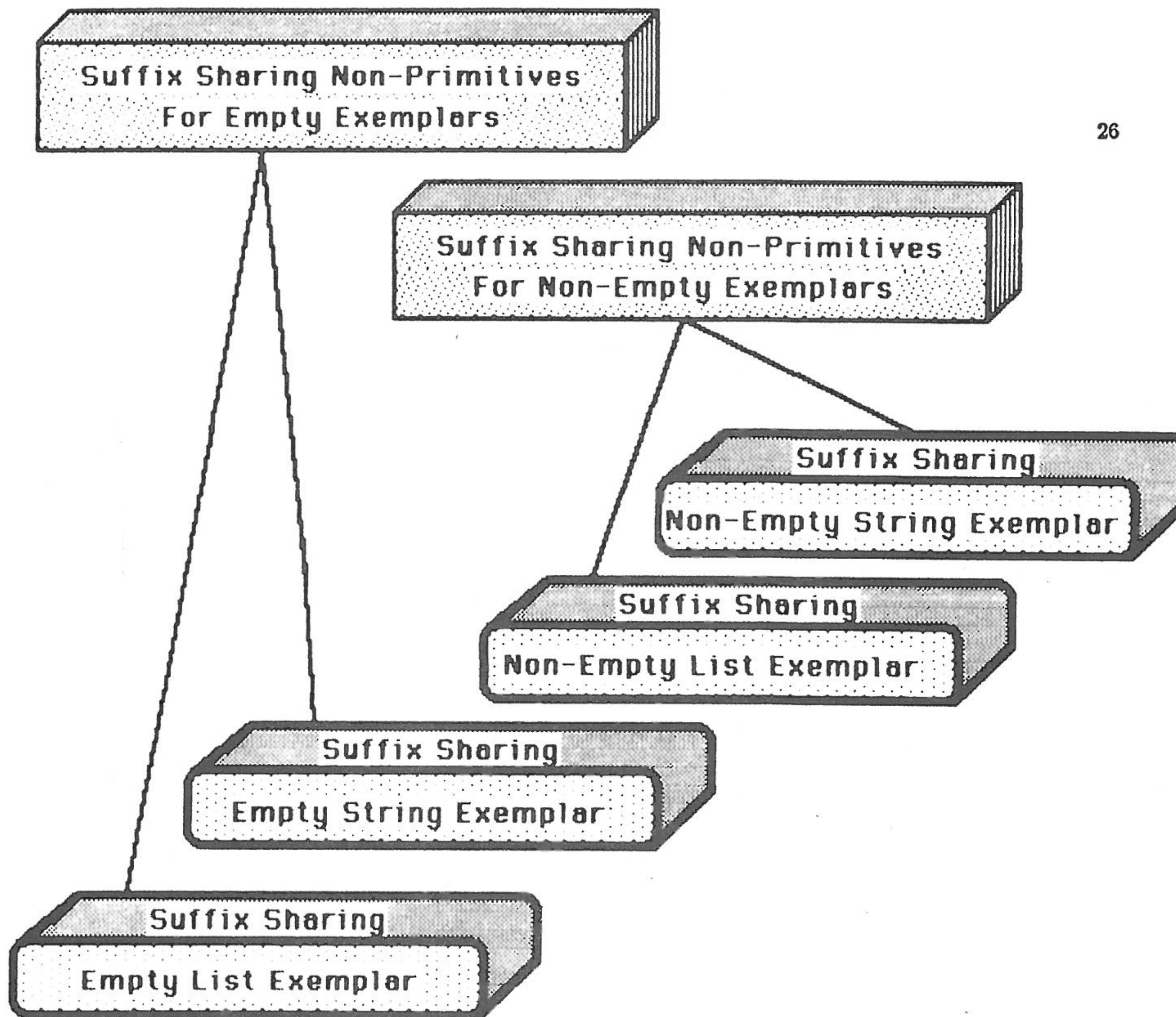
Carrying the design to completion should result in complementary organizations: a suffix-sharing design as shown in Figure 11 and a corresponding prefix-sharing version.

13 Multiple Inheritance

Multiple inheritance provides capabilities for sharing representation and operations between widely disparate data types.

Multiple inheritance permits sharing beyond the conventional tree-structured mechanism that permits subordinates to share the representation and operations of superordinates. It is supported by several programming languages and systems including Lisp[15], Loops[4], the Traits system[20], and extensions of Smalltalk[13]. The mechanism is ideal for constructing complex objects out of simpler components and providing the component operations to the whole. To use the Traits terminology, an object can be constructed by assembling a number of useful traits – each corresponding to a specific property of the object such as colour, position (two or three-dimensional), velocity, circumscribing envelope, ... Creating a new data type such as a plane for use in a video game is a matter of assembling the requisite properties and providing the missing components, both at the representation and operation level. For this example, a special set of icons and operations for displaying one of them – the choice depending perhaps on the direction of motion.

This notion of multiple inheritance plays the role of a selection mechanism that permits choosing from a large pool of properties. It is an alternative realization of a selection language for



A Complete Design for Suffix Sharing Varieties of Lists and Strings

Figure 11

specifying special varieties.

14 Conclusions

Smalltalk already provides effective support for programming-in-the-large but additional mechanisms and features are needed to

provide enhanced capabilities.

To summarize, designing data type communities requires attention to and use of facilities that programming-in-the-small need not concern itself with:

- *Ensuring that the data type name space remain small.*
- *Distinguishing between different data types and their varieties.*
- *Allowing data types to play a fundamental role as managers of their varieties.*
- *Distinguishing between class hierarchies and operation hierarchies.*
- *Partitioning the user hierarchy (i.e. classes) from the implementer hierarchy (i.e. varieties).*
- *Distinguishing carefully between representation accessing primitives and representation independent nonprimitives.*
- *Isolating and distinguishing between special exemplars that embody the gist of the data types.*

Our aim was to bring into focus some of the problems and issues that should be addressed when designing data types for integration into communities of data types. No programming language yet exists that directly supports these notions. Nevertheless, they serve as a modest beginning that can be pursued in two directions: (1) discovering and elucidating more powerful and more useful design methodologies and (2) designing and extending better facilities for programming in communities.

15 References

1. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, Mass., 1983.
2. Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, Mass., 1984.
3. Krasner, G., *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, Reading Mass., 1983.
4. Bobrow, D.G., and Stefik, M.J., *The LOOPS Manual (Preliminary Version)*, Knowledge-based VLSI Design Group Technical Report, KB-VLSI-81-13, Stanford University, August 1984.
5. Byrd, R.J., Smith, S.E. and de Jong, S.P., *An Actor-Based Programming System*, ACM SIGOA Conference on Office Information Systems, Univ. of Philadelphia, June 21-23, 1982.
6. Laff, M.R., *Smallworld - An Object-Based Programming System.* IBM Research Report RC-9022, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981.
7. Lieberman, H., *A Preview of ACT 1.* MIT AI Laboratory Memo No. 625, June 1981.
8. Lieberman, H., *Thinking About Lots of Things at Once Without Getting Confused - Parallelism in Act 1*, MIT AI Laboratory Memo No. 626, May 1981.

9. Green, M. and Philp, P., *The Use of Object-oriented Languages in Graphics Programming*, Proceedings of NCGA Graphics Interface 1982 Conference, Toronto, 1982.

10. Klahr, P., McArthur, D., and Narian, S., *SWIRL: An Object-Oriented Air Battle Simulator.* Proc. AAAI-82, Pittsburgh, August, 1982.

11. Reynolds, C.W., *Computer Animation with Scripts and Actors.* Proceedings of ACM SIGGRAPH Conference, July 1982.

12. Thalmann, D., Magnenat-Thalmann, N., *Actor and Camera Data Types in Computer Animation.* Proc. Graphcs Interface 83, Edmonton, Canada.

13. Borning, A. and Ingalls, D.H., *Multiple Inheritance in Smalltalk-80*, Proceedings of the AAAI Conference, Pittsburgh, Aug. 1982.

14. Cannon, H.I., *Flavors*, Technical Report, MIT Artificial Intelligence Lab., 1980.

15. Weinreb, D., Moon, D., *Flavours - Message-passing in the Lisp Machine.* MIT AI Memo No. 602, Nov. 1980.

16. Shapiro, E.Y. and Takeuchi, A., *Object Oriented Programming in Concurrent Prolog*, New Generation Computing, OHMSHA LTD and Springer-Verlag, Vol. 1, 1983, pp. 25-48.

17. Zaniolo, C., *Object-oriented Programming in Prolog*, 1984 International Symposium on Logic Programming, New Jersey, Feb. 1984, p. 265-271.

18. DeRemer F., and Kron, H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, SE-2, June 1976, pp 80-86.

19. McKeeman, W.M., Horning, J.J., and Wortman, D.B., *A Compiler Generator*, Prentice-Hall 1970.

20. Curry, B., Baer, L., Lipkie, D., and Lee, B., *Traits: An Approach to Multiple-Inheritance Inheritance Subclassing*, Proceedings ACM SIGOA Conference on Office Information Systems, published as ACM SIGOA Newsletter Vol. 3, Nos. 1 and 2, 1982.