DESIGNING FAMILIES OF DATA TYPES USING EXEMPLARS

Wilf R. LaLonde

SCS-TR-108

February 1987

School of Computer Science
Carleton University
Ottawa, Ontario
CANADA K1S 5B6

*This research was supported by NSERC, DREA and DREO.

Designing Families of Data Types Using Exemplars

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 expanding community of users is gaining experience with the object-oriented paradigm. Since designing classes is often viewed as synonymous with designing abstract data types, there is every expectation that users experienced with abstract data type design will adapt easily to designing classes. Unfortunately, there is a growing awareness that the reality is contrary to the expectation; i.e., data type designing in a system with inheritance is far more difficult than traditional books on data structures and data types have suggested. In object-oriented systems, designing is sometimes (but only rarely) concerned with the development of individual data types. It is more often concerned with the development of highly integrated families of data types and more generally communities of totally different but cohesive collections of data types that support specific applications; e.g., a compiler to pick a well-studied example.

Designing data types in isolation is fundamentally different from designing them for integration into communities of data types especially when inheritance is a fundamental issue. We are concerned with the design of families of data types as opposed to individual data types; i.e., on the issues that arise when the focus is intermediate between the design of individual data types and communities of data types. We argue there there is a need for familial classes to serve as an intermediary between users and the members of a family. We also argue that class-based systems provide inadequate facilities for the task and that exemplar-based systems are required. In addition, exemplar-based systems provide new design dimensions not provided by the latter. The important issues are presented under the guise of designing a family of List data types.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques -- modules and interfaces, software libraries; D.2.6 [**Software Engineering**]: Programming Environments; D.2.m [**Software Engineering**]: Miscellaneous -- reusable software; D.3.3 [**Programming Languages**]: Language Constructs -- abstract data types, data types and structures, modules, packages; D.3.4 [**Programming Languages**]: Processors -- compilers, interpreters, run-time environments; E.2 [**Data Structures**]: Lists.

General Terms: Algorithms, Design, Languages, Performance, Object-oriented systems, inheritance, classes, exemplars, prototypes, communities of data types, families of data types, exemplary programming, programming-in-the-large.

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 [9,10,13], partly to the ongoing research into object-oriented languages [1,4,14,17,18,21,26], partly to the novel applications for the paradigm [11,12,23,29], and partly to the increased emergence of object-oriented facilities in other well-known languages: Lisp [2,5,33], extensions of C [6,27], Pascal [28], Forth [8] and Prolog [24,32,34].

The small but growing community of users is slowly gaining experience with object-oriented software. While programming in a traditional language can begin with the design of a main program followed by a top-down decomposition, programming in an object-oriented language like Smalltalk is fundamentally different. No main program is ever written; instead, the problem must be partitioned into a collection of related but independent classes that together model the application. Although design is concerned with many separate issues like concurrency, extensibility, and simplicity that transcends the final product, in the end, the result is materialized in a collection of inter-related classes or data types. In a totally objectified language like Smalltalk, it may sound like an exaggeration to say that "all programming in the paradigm is concerned with designing and implementing data types" but the fact of the matter is that the maxim is a good approximation of reality. Because of this viewpoint, we might expect users with experience in designing abstract data types to easily switch over to the object-oriented languages. Unfortunately, our own experience at teaching undergraduates and graduates to design and program in the paradigm [22] or to rewrite existing programs written in Pascal, C, or Lisp fails to support the claim. In fact, there is a growing awareness that data type designing is far more difficult than traditional books on data structures and data types have suggested. This is most understood by 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 design 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" [7] where the focus is on communities of related data types and their interactions, it is clear that the "art" is far from understood and that existing programming languages are far from providing adequate support. In fact, the notion that the design of families of data types is a programming-in-the-large issue is not widely accepted. In a language that supports data types without inheritance, we would agree. With inheritance, however, designs that begin simply with isolated classes very soon evolve into complex families of related but slightly different classes. In order to share representations and operations, the classes are organized along carefully designed hierarchies. It is no longer possible as a user or an implementer to understand, use, extend, or

change a specific class as if it existed in isolation from the others. We believe that the inter-class relationships and their requirements for a more global view is a programming-in-the-large issue and a simplistic explanation for the observation that designing classes in a system supporting inheritance is fundamentally more complex than the corresponding task in a system without inheritance.

A **community** of data types could be defined as a collection of data types that support a specific application. For instance, a compiler as an application consists of many different data types integrated and designed specifically to support the compilation task. A robotics control application would have a similarly large and complex collection of data types -- a robotics community that might include data types for three dimensional coordinate manipulation, sensor detection, and effector control. Intermediate between the notion of a single data type and a community of data types is the notion of a **family** of data types, a highly integrated collection of similar but nevertheless different data types that reflect different space/time tradeoffs, different usage contexts, and different operations (with more similarities than differences). A family is a community of data types in which the members are closely related; distinct members are either specializations (generalizations) or variations of each other. Our goal is not to precisely define the boundary between families and communities but to highlight that there is a great difference between designing individual data types, designing families of data types, and designing communities of data types. Programming-in-the-small and in-the-large are clearly associated with the first and last areas. Designing families is a middle ground that has aspects of both. Additionally, designs must ultimately be realized on specific object-oriented systems. The inheritance mechanisms provided greatly influence the result.

Most object-oriented systems are **class-based** in the sense that the representation and operations associated with the instances are attached to the class. Smalltalk, in particular, is class-based. 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. Instead of attaching the representation and operations to the class, it is attached to the exemplar. Research supporting the increased power and flexibility of **exemplar-based** systems include [15,16,19]. The number of exemplar-based systems is small: Act 1 [17,18], Smallworld [14], Thinglab [3], and Actra [16,30]. The latter in particular is an exemplar-based Smalltalk.

On the surface, the differences between class-based and exemplar-based systems seem minor. However, the consequences are profound. First, inheritance in an exemplar-based system proceeds along a chain of instances (instance inheritance) while still permitting class inheritance (classes are instances of class objects). Second, several exemplars can be associated with the same class enabling the notion of multiple-representations to be supported.

In this paper, we illustrate how families of data types can be designed in an exemplar-based system. We also highlight the additional power of the exemplar-based systems and indicate how they can be used to alleviate and in some cases eliminate problems that are inherent to the class-based systems. We also introduce the concept of a familial class that serves as a intermediary between users and the members of the family. A familial class provides an interface to a collection of highly related data types much like an abstract data type provides an interface to the more detailed and more complex implementation.

2 Motivation

Using, modifying, and extending a rich object-oriented environment is extremely complicated.

Exploratory programming in a rich object-oriented system such as supported by the Smalltalk environment appears to be straighforward at first glance. However, difficulties arise as soon as serious programming efforts endeavour to use, modify, and extend families of related data types. For instance, novices desiring to become expert level programmers must become intimately familiar with the family of Collection data types -- there are currently 26 subclasses. In some cases, classes that have no logical relationship are physically related because it is convenient to inherit the representation; e.g., Dictionary is a subclass of Set for that reason even though logically the two are not related. In some cases, operations permitted on instances of a class are not permitted of a specialization; e.g. OrderedCollection permits **at:put:** but not the specialization SortedCollection. In general, the data type families tend to be organized in a manner that makes more sense to the implementers than it does to users of the classes.

Another family that should be mastered if special purpose interactive windows are to be designed (one of the attractions of Smalltalk) is the set of classes associated with View (22 subclasses) and Controller (28 subclasses). Because of the complexity, most people generally abandon the goal of understanding this family after a few days of browsing. Clearly, designing and using such densely connected families is not the same as designing a specific data type. In the long run, more powerful design tools and methodologies along with better investigation tools for users will be needed to support the programming process. Before we can do that however, we need a better understanding of the sources of the complexity and a catalogue of useful dimensions along which to organize this complexity.

Some of the complexity arises from (1) allowing the name space to grow in an unstructured and uncontrolled way and (2) failing to introduce a useful distinction between users (who need to know how to create and manipulate instances of the classes and little else) and implementers (who need to know much more).

One approach to managing and controlling these problems is to generalize inheritance from one hierarchy to two; i.e., to provide both an instance hierarchy and a class hierarchy as in the exemplar-based

systems. The class hierarchy is used for describing the logical relationships between classes and the instance hierarchy is used as an implementation vehicle for code and representation sharing. As we will see, this disinction cannot be separated in a class-based system. As a result, compromises with unresolvable deficiencies must be introduced.

The additional flexibility of exemplar-based systems permits the class hierarchy to be viewed principally as the **user's dimension** and the instance hierarchy as the **implementer's dimension**. The existing family of 26 Smalltalk Collection classes can be duplicated in such a system by defining 26 class exemplars and 26 instance exemplars; e.g., a class exemplar called Set and an instance exemplar called aSet, a class exemplar called Array and an instance exemplar called anArray, etc. The instance hierarchy is an exact mirror of the class hierarchy; i.e., the instance hierarchy from anArray to anObject duplicates the class hierarchy from Array to Object. On the surface, it appears that the exemplar-based approach is twice as complex as the class-based version because twice as much effort is required to realize a design. In reality, they are quite similar. In a class-based system, the creation of a class automatically causes the creation of a corresponding metaclass. The Array class, for example, contains the instance methods (operations) and representation; the Array metaclass contains the class methods and representation. Hence there are 26 Collection classes and 26 Collection metaclass in a class-based system.

The real difference between the two is that the class-based systems force the class and metaclass hierarchies to parallel each other. The exemplar-based systems do not; they also have no need for metaclasses. Because the class and instance hierarchies are decoupled, there is no need for a direct relationship between the two. Consequently, users can be provided with a view of the data types that is intuitive and logical. The implementation, on the other hand, is free to pursue any trick-of-the-trade to provide the desired space/time efficiencies as long as the user's view is maintained.

There is another fundamentally new capability provided by the exemplar-based approach. The designer need not provide a corresponding class exemplar for each instance exemplar. For example, the existing Collection family could be re-designed with only one class exemplar and 26 instance exemplars. Collection in this case would be more like a parameterized class with 26 possible parameterizations, one for each of the 26 possible behaviours engendered by the corresponding instance exemplars. Alternatively, there could be a compromise that distinguishes only the more important classes such as Array, OrderedCollection, Bag, String, and Symbol. Array, for instance, could be parameterized to provide either a dictionary or identity dictionary (arbitrary subscripts), standard array (integer subscripts), byte array (elements that are very small integers), etc.

To explore these design issues and illustrate the complexity that can arise when designing a family of data types, we will consider a design for a family that complements the existing Collection data types; i.e., a family of List data types in the tradition of Lisp. Consequently, the rest of the paper assumes the reader is both familiar with Lisp lists and Smalltalk. We also assume that an exemplar-based Smalltalk

such as in Actra [16,30] is available; the differences with a standard Smalltalk will be evident to those who already know Smalltalk or the basic principles of object-oriented systems with inheritance.

Lisp lists are an example of a **sharable** data type; i.e., a data type in which part of an instance is itself a separate instance of the same type (by design) and capable of independent manipulation. One of the main reasons for the power and efficiency of Lisp's list manipulation capabilities is the fact that its lists are sharable. By contrast, the Smalltalk Collection classes can be described as **non-sharable** lists. No part of an existing collection can itself be manipulated as a collection of the same type by design although this could happen by accident if one of the elements happened to be a collection. Actually, there exists a LinkedList class that provides an approximation to Lisp lists but it is much too specialized. It is used by the system for managing processes and is ignored by most users. By contrast with other collection classes that permit arbitrary objects to be inserted, these linked lists permit only instances of Link to be inserted; Links can be specialized to contain arbitrary objects. The two-level design is cumbersome by comparison with Lisp lists.

The exposition proceeds in the manner that typical families are evolved; i.e., a specific list data type is first designed by focusing on a useful set of operations and the sharing model previously mentioned and positioning the data type in a logical hierarchy. A conventional class-based solution is then proposed and evaluated. An alternative exemplar-based solution is next shown to be superior. Then we consider a complementary sharing model, a corresponding data type, and an implementation. The emphasis in the latter two solutions is the implementation hierarchy. Finally, a third data type based on yet another sharing model is proposed and implemented. Again, a distinct implementation technique is used. Then we come back to the logical hierarchy and consider alternative designs and their impact on the physical hierarchy.

We emphasize that it is not a detailed design that we are focusing on but the detailed issues that could and should be addressed when attempting to construct such a design. By highlighting these issues, we hope to suggest directions for the design of future object-oriented systems.

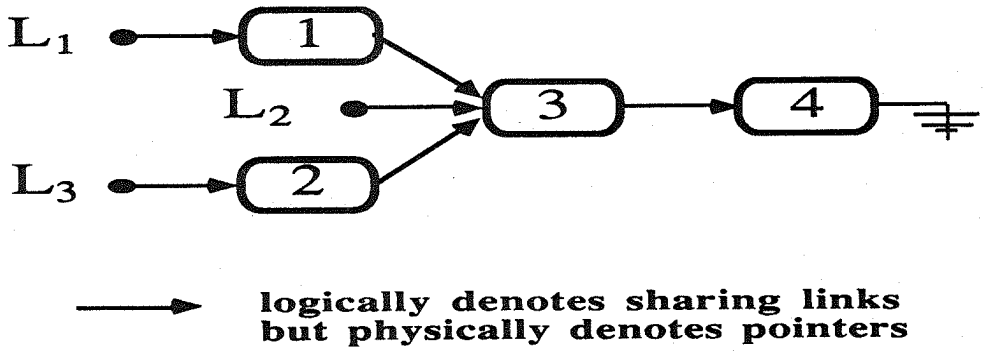
3 Designing the List Operations

A preliminary specification that focuses on a few key properties 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 the messages be designed first (i.e., the specification) with the representation and the implementation of the methods last. We focus our attention on a design that meets two key requirements:

(1) the lists must be sharable, and
(2) multiple empty lists must be provided.

Sharability is a user level property not to be confused with a corresponding implementation level property that is an implementation trick for conserving space. To properly use lists, users must understand how sharing is achieved. If we adopt the existing Lisp design, sharing is supported in a very restricted manner as shown in Figure 1.



The Lisp Sharing Model
Figure 1

For example, L_1 is a list containing elements 1, 3, and 4 with the last two elements shared by L_2; similarly, L_3 contains 2, 3, and 4 with the last two elements also shared by L_2. In this model, sharing occurs on the right; i.e., the tail or "rest" of the list. Other models will be considered later.

The requirement for multiple empty lists (as opposed to having just one that is shared by all non-empty lists) is imposed to eliminate special-cased semantics that would otherwise result for certain specialized operations (see 3-Lisp [25]). For example, operation **copy** should return a new list independent of whether or not it is empty. Operation **destructiveAppend:** should modify the receiver by appending the additional elements. This cannot be done for a receiver that is an empty list if only one empty list is provided by the system.

The end-result of the design is a List data type that emphasizes non-copying semantics for operations that match well with the given restrictions on sharability and copying semantics otherwise. A summary of the more important and illustrative operations is shown in Table 1. Note that an operations invoked as "aList **sublist:** 1 **to:** 5" is referred to as "**sublist:to:**" in Smalltalk.

Instance Operations

isEmpty (aList) => Boolean: Is the list empty?
size (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. (**shared**)
last (aList) => Object: The last element of the list. (**shared**)
element: (aList, aPosition) => Object: The element of the list at the specified position. (**shared**)

rest (aList) => List: The remaining (all but first) elements of the list. (**shared**)
allButLast (aList) => List: The remaining (all but last) elements of the list. (**copied**)

prefix: (aList, aNewSize) => List: An initial portion of the list. (**copied**)
suffix: (aList, aNewSize) => List: A final portion of the list. (**shared**)
sublist:for: (aList, aStart, aNewSize) => List: A subportion of the list. (**copied**)
sublist:to: (aList, aStart, anEnd) => List: A subportion of the list. (**copied**)

precede: (aList, anObject) => List: A new list with anObject as the first element. (aList **shared**)
follow: (aList, anObject) => List: A new list with anObject as the last element. (aList **copied**)
append: (aList, anotherList) => List: A new list obtained by appending the two. (anotherList **shared**)

copy (aList) => List: A new list equal to aList. (corresponding elements **shared**)

replaceFirst: (aList, anObject) => List: Changes the first element of the list. (**shared**)
replaceRest: (aList, anotherList) => List: Changes the remainder of the list. (**shared**)
destructiveAppend: (aList, anotherList) => List: Adds anotherList to the end of aList. (anotherList **shared**)

do: (aList, aBlock) => Nothing: Invokes the one-parameter block (function) on each list element.
collect: (aList, aBlock) => List: Invokes the one-parameter block (function) on each list element and collect the answers into a new list.

Class Operations

empty (theListClass) => List: A new empty list.

Table 1: An Illustrative Subset of the List Operations

We have not tried to be complete but we have tried to be illustrative. For instance, operations ~== (not of ==) and ~= (not of =) are omitted; so are operations for checking the type of an instance, for converting to other types, for reading/writing lists, etc. Additional destructive operations like **replaceLast** and **replaceAllButLast** and enumeration operations beyond **do** and **collect** could also be provided. Because the first parameter is distinguished as the receiver in Smalltalk, all instance operations start with a list instance as the first parameter and all class operations with a list class. More powerful selection mechanisms that choose the receiver based on the type of all parameters do exist [2] for other languages but they are not yet in widespread use. This has led, for example, to a different choice of name for the Lisp operation **cons** which has been renamed **precede** above.

Note that we have specifically differentiated between those operations that share and those that do not. We will come back to this point later, but for the moment, consider **precede** and **follow**. The particular sharability model used permits **precede** to be designed in such a manner that the new list constructed simply encapsulates the old list and the new object. Thus L_1 in Figure 1 can be obtained by executing **precede** (L_3, 1) **without any side-effects** on the parameters. Since **follow** is a complementary operation, it too should have no side-effects. It is therefore not allowable for **follow** (L_3, 4) to destructively add 4 to the end of L_3 since L_1 and L_2 would also have been modified as a side-effect. There is no recourse but to copy L_3; i.e., to construct a new list containing all of the elements of L_3 including 4. An operation such as **destructiveFollow** should also be provided for completeness but it must be clearly differentiated from the version without side-effects.

The resulting design incorporates many ideas that are the proper concern of designing-in-the-small. In particular, operation efficiency (emphasizing sharability whenever possible), semantics that are not special-cased (sublist could have been designed to "share" when a suffix is requested and copy otherwise), name design (in Lisp **precede** is called **cons** but the parameters are reversed; in Smalltalk, the receiver (the first parameter) must be a list), symmetric extension (if there is a **first**, there should also be a **last**, ...), integration (Smalltalk provides == for all objects, it must be provided for lists too), compability (list elements are shared just like elements of collections; this is also an efficiency issue), convenience (two versions of **sublist** are provided, one requiring a start and an end point; the other, a start and a size).

Since designing-in-the-small is not a focus of this paper, we have not tried to illustrate the process used to obtain the above design. For data types that are well-known, it is clearly possible to produce new designs that are improvements over previous ones using creative but conventional offline approaches. The design of new but unknown data types does not often follow the same methodology. In an interactive system like Smalltalk, new data types are typically introduced "by need" and their designs polished by evolving them as new demands and requirements are imposed on the data types. The resulting designs are often dramatically different from the initial simplistic ones that started the process.

4 Designing The Logical Hierarchy

The difference between a logical hierarchy and an implementation hierarchy can cause confusion especially when they have different inheritance directions

Although designing a new data type and placing it in a hierarchy is not typically done independently

in Smalltalk, they are two separate tasks and they could be done independently. For instance, after a type has been designed, we could review the existing hierarchy and position it so as to minimize the implementation effort. With care, a large subset of the designed operations would be inherited to eliminate part of the implementation effort. We would also have to analyse the additional operations that come "for free" because of inheritance to ensure they all made sense for the new data type. In some cases, extra effort may be required to override the operations by re-defining them in some appropriate way.

What is not widely known, however, is the fact that the hierarchy is being used for two distinct and sometimes conflicting purposes. It is being used to describe both the logical relationships between the classes and the physical relationships that permit code and representations to be shared. We disgress for a moment to expand on this point.

As users of an object-oriented system, it is important to understand the relationships between the classes and the operations relevant to the instances without having to resort to studying the implementation. When a class is described as a **specialization** of another class, we expect the specialization to have certain properties.

1. It typically has more operations for its instances; e.g., specializing Object to Integer introduces operations + and -. In unusual cases, operations may be removed but this is not typical.
2. It may have a special-purpose representation for more efficient storage and/or access.
3. It should be possible to treat the instances of the specialization as an instance of the more general type.

Implementers of new classes must assume a dual responsibility. Since they are designing classes for the user community, they must attempt to have them satisfy the above properties. They must also realize the intended behaviour as efficiently as possible. Problems arise as soon as the two concerns begin to conflict.

For an example, consider classes Set and Bag in Smalltalk (a bag is a set that allows duplicates). Both have the same set-like operations but they have slightly different semantics for the insert operation. Bags insert all submissions. Sets permit the insertion only if no duplicate exists; i.e., inserting a duplicate has no effect. Logically, every set is a bag but not every bag is a set. Hence Set should be a specialization of Bag. A straightforward implementation that maintains this view would have Set inherit from Bag. If Bag maintains counts for the number of duplicates, this count could be restricted to 1 for Sets. A better design, however, would likely remove the counts altogether -- a notion not possible with the class model unless the logical hierarchy is changed. The Smalltalk solution was to make both Set and Bag subclasses of Collection.

The reason for the conflict is that two hierarchies are involved: a class hierarchy that describes a Set

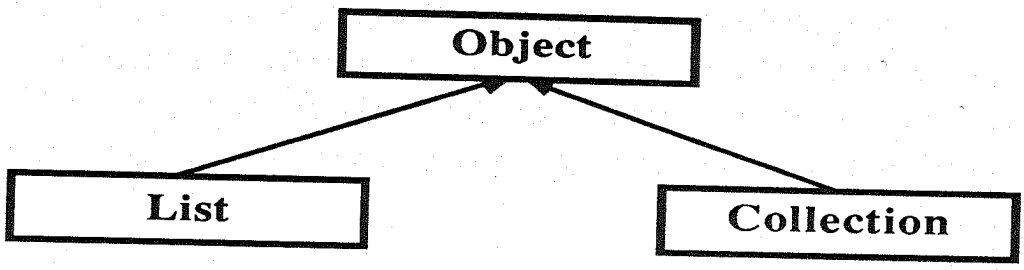
as a special case of a Bag and an instance hierarchy that implements a set instance as a special case of a collection instance. If the instance hierarchy is forced to match the class hierarchy, we end up with a meaningful logical relationship but an inefficient implementation. Conversely, if the class hierarchy is forced to match the instance hierarchy, we end up with a relationship that is logically wrong but nevertheless space efficient.

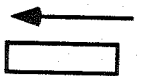
The above is not an isolated example. When the logical hierarchy (the class hierarchy) and the physical implementation hierarchy (the instance hierarchy) are forced to be the same, the physical hierarchy usually wins out. For another example, Dictionaries in Smalltalk are an obvious generalization of Arrays in which the indices are arbitrary objects. Logically, Array should be a subclass of Dictionary and Dictionary should be a subclass of Object. Currently, however, Dictionary is a subclass of Set simply because it happens to be using the Set representation. This time the class hierarchy was forced to match the instance hierarchy because it was convenient to inherit a specific existing representation. Since Smalltalk is class based, a future change in the representation could entail a change in the relationship between classes.

The data type community has long advocated a separation between what a data type does (the operations and their semantics) and its implementation (the representation and coding). This separation is provided by exemplar-based object-oriented systems.

In an exemplar-based system, class Set can be made to inherit from class Bag to establish the logical relationship between the two (the user's view). On the other hand, the set's instance exemplar can be made to inherit from the collection's instance exemplar to maintain the existing implementation (the implementer's view). The same approach could be used for Arrays and Dictionaries.

Returning from our digression, we can continue our List design with the knowledge that interactions between the user's view and the implementer's view are a possibility. However, the potential conflict is completely ignored as we focus entirely on the design of the logical hierarachy. For simplicity, we settle on the design of Figure 2. There is no particular design issue to highlight at this stage but we need a simple reference point for later discussions.





Part of a Class Hierarchy Containing Lists
Figure 2

The design clearly partitions object **containers** into two classes: lists providing sharable containers and collections providing non-sharable containers; i.e., users think of lists as a sharable version of collections; they also think of collections as a non-sharable version of lists. So far, Lists are degenerate as a family since there is only one member but Collection actually expands to 26 subclasses. A more complex design could be pursued that organizes one of the two container families as a special case of the other. Such inter-family relationships is a legitimate programming-in-the-large problem but we will not pursue it here. Instead, we will concern ourselves with interactions between the members (yet to be designed) of the List family.

5 Designing An Implementation Hierarchy With Classes

Designing an implementation hierarchy that conforms to the logical hierarchy does not always lead to the most efficient realization.

The hierarchy shown above clearly presents the user's view of List. The implementer's task is to choose a representation and a realization for the operations that maintains the semantics (a programming-in-the-small issue). At the same time, the implementer must consider the realization from the perspective of a hierarchy to minimize implementation effort. He must also take into account potential specializations that other designers might wish to introduce later and/or possible future reorganizations of the hierarchy to ensure that these may be done as painlessly as possible (a programming-in-the-large issue).

If a new specialization is to be introduced later that can profitably inherit from the version to be realized now, there must be a distinction between operations that are **non-primitive**; i.e., depend entirely on others and those that are **primitive**; i.e., a kernel set that is sufficient to support the

non-primitives. Primitives typically have direct access to the representation and/or provide a capability that serves as a base for the implementation of other operations. 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. Our only contribution in this respect is the observation (obvious in this context) that distinguishing primitives from non-primitives is of concern only to the implementer, not the user. Consequently, the distinction is not made in the logical hierarchy.

In Smalltalk, for example, class Magnitude was designed with such extensibility in mind. New subclasses of Magnitude need only provide an implementation of =, >, and **hash** (the primitives) to ensure that all other operations work with the new specialization; i.e., operations such as >=, <=, ~=, and **between:and:** are non-primitive. Although the idea is known, there are few examples using the methodology. When designing a new specialization of Number, for instance, it is not clear which operations are primitive since they have not been singled out.

If we consider the representation for our lists, it seems reasonably clear that it must include a **firstPart** and a **restPart**. If multiple empty lists are to be provided, some mechanism must be devised to differentiate empty lists from non-empty lists. One way might be to use an implementation trick: keep a non-list object in **restPart** to signal an empty list. A more expensive but less tricky solution would be to add a new component **listIsEmpty** that indicates whether or not this instance denotes an empty list. With either representation, it seems reasonably clear that non-destructive operations **isEmpty**, **first**, **rest**, and **precede** must be primitive along with destructive operations **replaceFirst** and **replaceRest**. Verifying that the others are non-primitive amounts to sketching their implementation in terms of the primitives (or other non-primitives). See operations **last**, **follow**, **prefix**, and **suffix** below for example. The implementation that follows uses the tricky solution and happens to match the logical hierarchy proposed in the previous section. We will change this later.

Lists - A Class-Based Implementation

```
class name                                  List
superclass                                  Object
instance variable names                     firstPart restPart

class methods

empty list creation

empty
    "We assume the list is empty if restPart is not a list"
    ↑self new "By default, restPart is nil; hence the list is empty"
```

instance methods

querying

```
isEmpty
    ↑(restPart isKindOf: List) not

size
    self isEmpty ifTrue: [↑0] ifFalse: [↑1 + self rest size]
```

left end operations

```
first
    self isEmpty ifTrue: [↑self error: 'first is not legal on empty lists'] ifFalse: [↑firstPart]

rest
    self isEmpty ifTrue: [↑self error: 'rest is not legal on empty lists'] ifFalse: [↑restPart]

precede: element
    "Ensures that first (precede (L, O)) == O and rest (precede (L, O)) == L"
    ↑(self new replaceFirst: element) replaceRest: self

prefix: newlength
    newlength < 0 ifTrue: [↑self error: 'illegal prefix size'].
    newlength = 0 ifTrue: [↑self class empty].
    ↑(self rest prefix: newlength - 1) precede: (self first)
```

right end operations

```
last
    self isEmpty
        ifTrue: [↑self error: 'last is not legal on empty lists']
        ifFalse: [self rest isEmpty ifTrue: [↑self first] ifFalse: [↑self rest last]]

allButLast
    self isEmpty
        ifTrue: [↑self error: 'allButLast is not legal on empty lists']
        ifFalse: [
            ↑self rest isEmpty
                ifTrue: [↑self class empty]
                ifFalse: [↑(self rest allButLast) precede: (self first)]]

follow: element
    self isEmpty
        ifTrue: [↑(self class empty) precede: element]
        ifFalse: [↑(self rest follow: element) precede: (self first)]

suffix: newlength
    | result size |
    result ← self. size ← self size.
    (newlength < 0) | (newlength > size) ifTrue: [↑self error: 'illegal suffix size'].
    1 to: (size - newlength) do: [:i | result ← result rest].
    ↑result
```

other manipulations

```
element: position
    position < 1 | self isEmpty ifTrue: [↑self error: 'illegal element position'].
    position = 1 ifTrue: [↑self first] ifFalse: [↑self rest element: position - 1]

append: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot append a non-list to a list'].
    self isEmpty
        ifTrue: [↑list]
        ifFalse: [↑(self rest append: list) precede: (self first)]

sublist: start for: size
    ↑(self suffix: self size - start + 1) prefix: size

sublist: start to: end
    ↑(self suffix: self size - start + 1) prefix: end - start + 1

copy
    self isEmpty
        ifTrue: [↑self class empty]
        ifFalse: [↑self rest copy precede: self first]
```

destructive modifications

```
replaceFirst: element
    self isEmpty
        ifTrue: [↑self error: 'replaceFirst is not legal on empty lists']
        ifFalse: [firstPart ← element. ↑self]

replaceRest: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot replace the rest of a list by a non-list'].
    self isEmpty
        ifTrue: [↑self error: 'replaceRest is not legal on empty lists']
        ifFalse: [restPart ← list. ↑self]

destructiveAppend: list
    | tail |
    (list isKindOf: List) ifFalse: [↑self error: 'cannot append a non-list'].
    tail ← self. [tail isEmpty] whileFalse: [tail ← tail rest].
    "Note: the traditional Smalltalk become: is a two-way (swapping) become"
    "X oneWayBecome: Y => Change all and only X-references to Y-references"
    tail oneWayBecome: list.
    ↑self
```

sequencing

```
do: block
    self isEmpty
        ifTrue: [↑self]
        ifFalse: [block value: self first. ↑self rest do: block]

collect: block
    self isEmpty
        ifTrue: [↑self class empty]
```

ifFalse: [↑(self **rest collect:** block) **precede:** (block **value:** self **first**)]

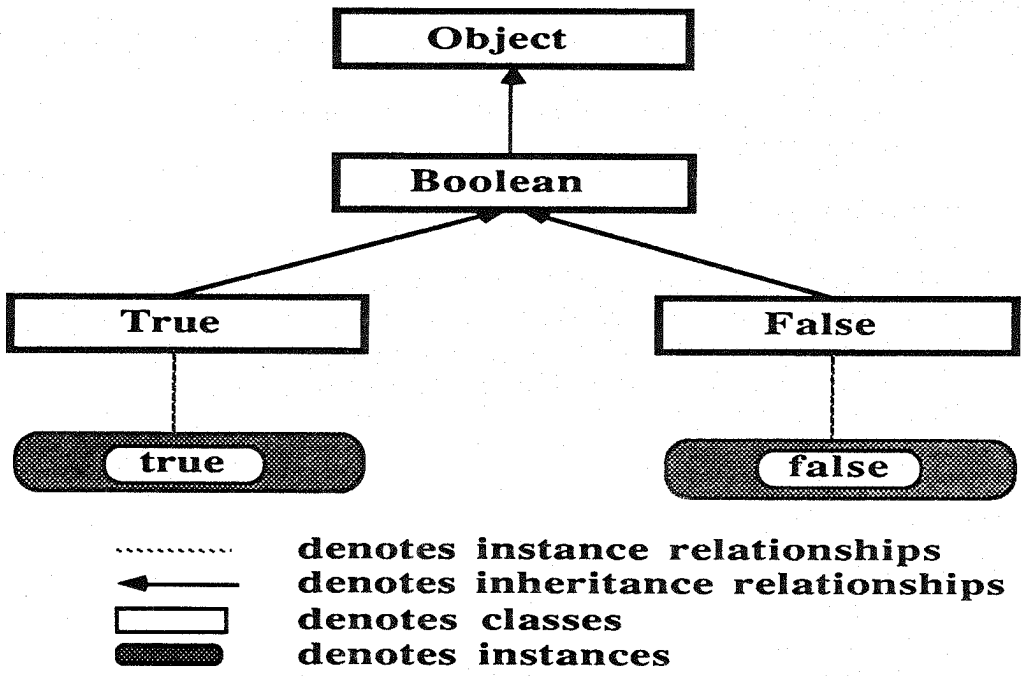
This realization is traditional but it is not the most time-efficient implementation. A quick glance at the implementation should make it quite clear that virtually all operations perform a test to differentiate between the empty and non-empty list cases. The test is pervasive and cannot be directly removed without a substantial design change. Yet it can be done but it requires an exemplar-based design.

6 Designing An Implementation Hierarchy With Exemplars

Designing instances with dictinct representations leads to more efficient implementations.

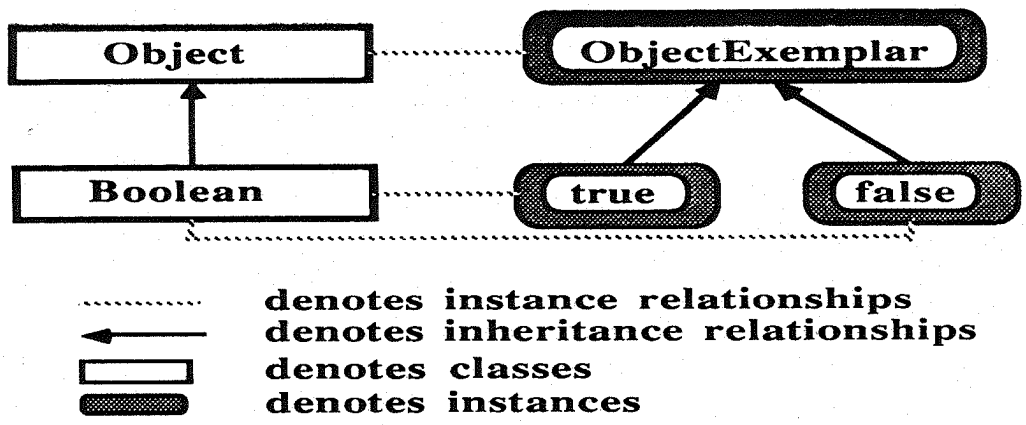
To obtain a time-efficient realization of the list data type, it is necessary to switch to an exemplar-based design with two representations: one for empty lists and another for non-empty lists. This exemplar-based design can either be simulated in a class-based system or provided directly in an exemplar-based system.

To illustrate the approach prior to providing the revised list design, consider the definition of a class Boolean with two instances **true** and **false**. Figure 3 presents a class-based design simulating exemplars while Figure 4 presents a more direct exemplar-based approach. The former design is actually used in Smalltalk.



A Class Based Design

Figure 3



An Exemplar Based Design

Figure 4

Both **true** and **false** have an empty representation (no instance variables). However, each instance is given a different implementation of the methods to take advantage of the run-time efficiency made possible. For illustration, the **and:** method for the two instances are shown below:

```
"for true"                          "for false"
and: anotherBoolean                 and: anotherBoolean
    ↑anotherBoolean                     ↑false
```

In a more conventional design, each boolean instance would need at least one field (or bit) for the representation; e.g., containing 1 for **true** and 0 for **false**. More important, this field must be explicitly tested and/or extracted to perform the boolean operations. Such explicit access is superfluous if the exemplar approach is used.

In a class-based system, the only way to implement **and:** for **true** differently from **and:** for **false** is to have them be members of distinct classes. Consequently, class True is defined specifically for its one instance **true** (similarly for class False and instance **false**). Although not particularly useful, it is possible to permit multiple instances of **true** and **false**.

In an exemplar-based system, a class exemplar for Boolean is created along with two instance exemplars: one for **true** and one for **false**. Class Boolean inherits from class Object whereas the instances **true** and **false** inherit from ObjectExemplar.

Looking ahead, it should be clear that implementing lists with simulated exemplars will result in a hierarchy that is different from the hierarchy originally designed for the user's perspective. On the other hand, the non-simulated approach can accommodate the logical hierarchy directly. Of course, it can easily be argued that the list family is so small at this stage that changing the user's perspective is a reasonable compromise. More substantial objections against the simulated approach will subsequently be presented.

```
class name                                  List
superclass                                  Object
class variable names                        "none"

methods

empty list creation

empty
    ↑EmptyListExemplar clone

non-empty list creation

nonEmpty
    ↑NonEmptyListExemplar clone


exemplar name                               EmptyListExemplar
superexemplar                               ObjectExemplar
class name                                  List
instance variable names                     "none"

methods

querying

isEmpty
    ↑true

size
    ↑0

left-side manipulations

first
    ↑self error: 'first is not legal on empty lists'

rest
    ↑self error: 'rest is not legal on empty lists'

precede: element
    ↑(self class nonEmpty replaceFirst: element) replaceRest: self

prefix: newlength
    newlength = 0 ifFalse: [↑self error: 'illegal prefix size'].
    ↑self class empty

right-side manipulations

last
```

```
    ↑self error: 'last is not legal on empty lists'

allButLast
    ↑self error: 'allButLast is not legal on empty lists'

follow: element
    ↑(self class empty) precede: element

suffix: newlength
    | list |
    newlength = 0 ifFalse: [↑self error: 'illegal suffix size'].
    "Must not manufacture a new empty list"
    list ← self.
    [list isEmpty] whileFalse: [list ← list rest].
    ↑list

other manipulations

element: position
    ↑self error: 'position outside bounds of list'

append: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot append a non-list to a list'].
    ↑list

sublist: start for: size
    size = 0 ifFalse: [↑self error: 'illegal sublist size'].
    ↑self class empty

sublist: start to: end
    (start >= (end + 1)) ifFalse: [↑self error: 'illegal sublist size'].
    ↑self class empty

copy
    ↑self class empty

destructive modifications

replaceFirst: element
    ↑self error: 'replaceFirst is not legal on empty lists'

replaceRest: list
    ↑self error: 'replaceRest is not legal on empty lists'

destructiveAppend: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot append a non-list'].
    "Note: the traditional Smalltalk become: is a two-way (swapping) become"
    "X oneWayBecome: Y => Change all and only X-references to Y-references"
    ↑self oneWayBecome: list

sequencing

do: block
    ↑self
```

```
collect: block
    ↑self class empty
```

exemplar name	NonEmptyListExemplar
superexemplar	ObjectExemplar
class name	List
instance variable names	firstPart restPart

methods

querying

```
isEmpty
    ↑false

size
    ↑1 + self rest size
```

left-side manipulations

```
first
    ↑firstPart

rest
    ↑restPart

precede: element
    ↑(self class nonEmpty replaceFirst: element) replaceRest: self

prefix: newlength
    newlength <= 0 ifTrue: [↑self class empty].
    ↑(self rest prefix: newlength - 1) precede: (self first)
```

right-side manipulations

```
last
    self rest isEmpty ifTrue: [↑self first] ifFalse: [↑self rest last]

allButLast
    self rest isEmpty
        ifTrue: [↑self class empty]
        ifFalse: [↑(self rest allButLast) precede: (self first)]

follow: element
    ↑(self rest follow: element) precede: (self first)

suffix: newlength
    | result size |
    result ← self. size ← self size.
    (newlength < 0) | (newlength > size) ifTrue: [↑self error: 'illegal suffix size'].
    1 to: (size - newlength) do: [:i | result ← result rest].
    ↑result
```

other manipulations

```
element: position
    position = 1 ifTrue: [↑self first].
    position < 1
        ifTrue: [↑self error: 'illegal element position']
        ifFalse: [↑self rest element: position - 1]

append: list
    ↑(self rest append: list) precede: (self first)

sublist: start for: size
    ↑(self suffix: self size - start + 1) prefix: size

sublist: start to: end
    ↑(self suffix: self size - start + 1) prefix: end - start + 1

copy
    ↑self rest copy precede: self first

destructive modifications

replaceFirst: element
    firstPart ← element. ↑self

replaceRest: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot replace the rest of a list by a non-list'].
    restPart ← list. ↑self

destructiveAppend: list
    self rest destructiveAppend: list. "The empty list at the right end must become the list"
    ↑self

sequencing

do: block
    block value: self first. self rest do: block

collect: block
    ↑(self rest collect: block) precede: (block value: self first)
```

Since the method lookup overhead is the same for both empty and non-empty lists (also the same for the more traditional implementation), removing the self test for distinguishing the two cases is clearly superior. This can be verified by considering an operation like **copy** on an unknown list; if it happens to be an empty list, a new empty list is returned without the self test required in the traditional implementation; if it happens to be a non-empty list, a new list is automatically constructed again without a self test that checks for a non-empty list. This multiple representation approach to designing data types is relatively unknown. The process is reminiscent of the partitioning process that Prolog programmers must deal with. For example, compare

"**append** (List1, List2, Result)"

```
"For an empty list"
append ([], List2, List2).

"For a non-empty list"
append ([FirstOfList1 | RestOfList1], List2, [FirstOfList1 | RestOfList1WithList2]) :-
    append (RestOfList1, List2, RestOfList1WithList2).
```

with

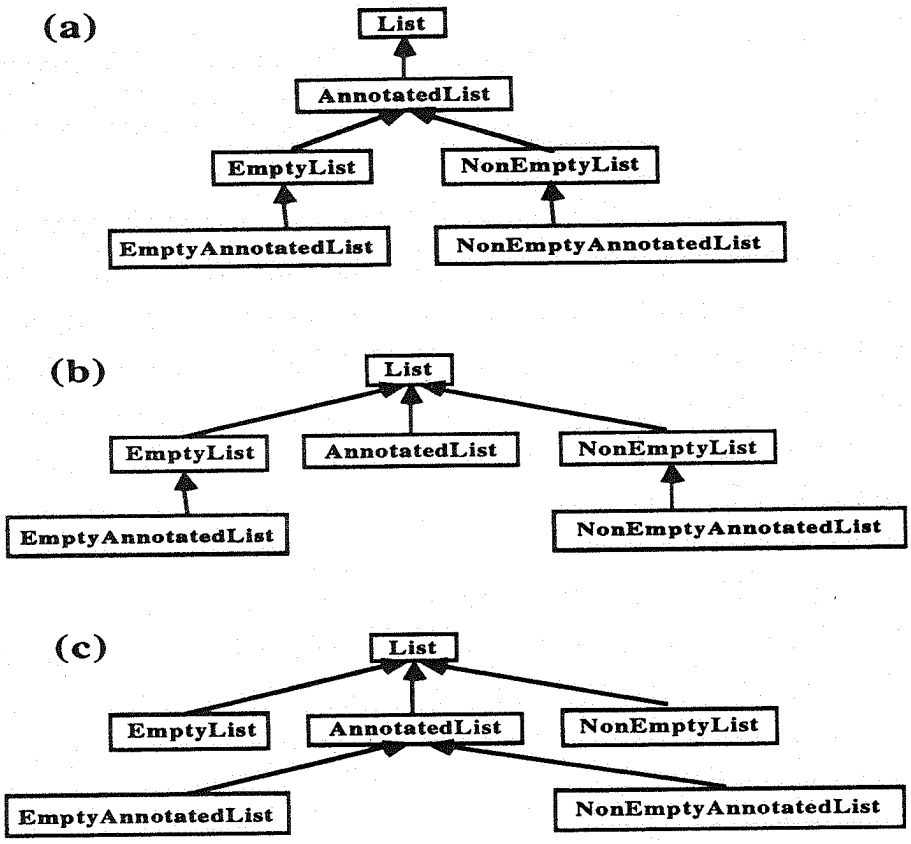
```
"For an empty list"
append: list
    (list isKindOf: List) ifFalse: [↑self error: 'cannot append a non-list to a list'].
    ↑list

"For a non-empty list"
append: list
    ↑(self rest append: list) precede: (self first)
```

7 Why Not Just Simulate The Exemplar-Based Design?

Simulating exemplars with classes is not as convenient as providing exemplars directly.

Although we presented a non-simulated exemplar-based solution above, we could have just as easily provided a simulated class-based version by designing NonEmptyList and EmptyList as specializations of List. This approach is fine as far as it goes but difficulties appear as soon as further specializations need to be introduced. Remember that we are not concerned solely with this particular realization but also potential extensions that other designers might wish to provide. For example, suppose a designer wished to introduce a new class, AnnotatedList say, as a specialization of List (it has additional special purpose instance variables) with two instance exemplars for empty and non-empty annotated lists respectively. Such lists might be used for encoding a list representation of a program with the additional annotation providing column positions permitting the original source to be reconstructed. Even comments could be accommodated as a special comment annotation.

To integrate AnnotatedList with the simulated class-based version of List, we expect NonEmptyAnnotatedList to be below NonEmptyList in the inheritance hierarchy (also EmptyAnnotatedList below EmptyList) in order to inherit the many methods provided. However, there is no convenient location in the hierarchy for class AnnotatedList.



Simulating Exemplars With Classes
Figure 5

Several approaches are possible (see Figure 5). Each has its own deficiencies. Placing AnnotatedList between List and its two specializations (Figure 5a) causes standard lists to be erroneously viewed as annotated lists. Placing it below List and beside its two specializations (Figure 5b) has the implication that EmptyAnnotatedList, for example, is not an AnnotatedList. The only remaining solution is to place AnnotatedList below List (Figure 5c) with EmptyAnnotatedList and NonEmptyAnnotatedList below AnnotatedList. However, this is unacceptable because all list instance methods must be duplicated. The only reasonable solution is to use either case (b) with method **isKindOf:** for EmptyAnnotatedList and NonEmptyAnnotatedList modified to return **true** if its parameter is AnnotatedList (a compromise that makes the incorrect physical hierarchy appear logically correct) or use multiple inheritance (case (b) modified so that EmptyAnnotatedList and NonEmptyAnnotatedList additionally inherit from AnnotatedList).

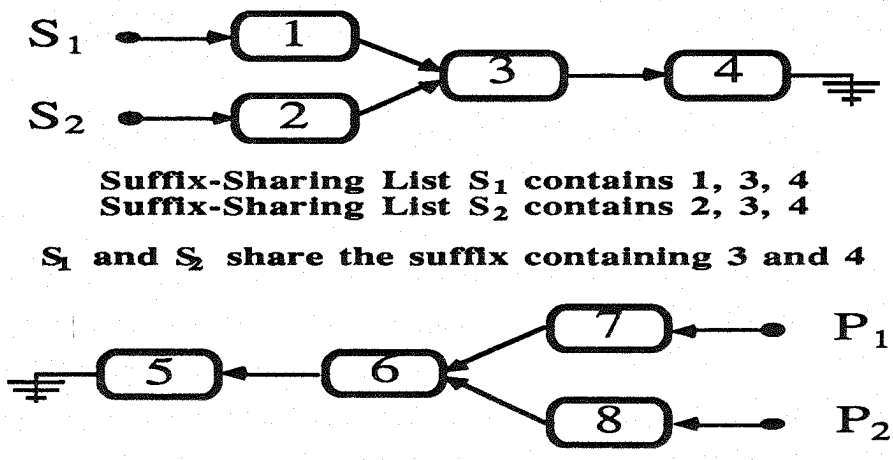
To summarize, exemplars can be simulated with the class-based systems to provide more efficient response but multiple levels of exemplars become increasingly problematical. The issue does not arise in exemplar-based systems sinces classes and instances form separate hierarchies.

8 Extending The Family

The notion that implementing a new type causes old types to have to be generalized or renamed is unique to systems with inheritance

If we briefly review the list operations, it is easy to see that the lists can be extended easily on the left (using **precede:**) but not on the right (**follow:** is expensive). Moreover, the first element (using **first**) is easily extracted but the last element (using **last**) can only be obtained by sequencing through all preceding elements. Similarly, **rest** is an inexpensive operation but **allButLast** must construct a new list with the required elements. This asymmetry is a direct result of the particular sharing model used in the design. This model can be made more evident by choosing a more expressive name for the lists; e.g., **suffix-sharing** lists.

This new emphasis makes it clear that a complementary class of lists could be designed; i.e., **prefix-sharing** lists. Such lists would be constructed easily by adding to the right (using **follow:**) and traversed easily from right to left (using **last** and **allButLast**). For this model of sharing, it is **first**, **rest**, and **precede:** that are expensive. Figure 6 contrasts these two kinds of sharing.



Suffix-Sharing List S_1 contains 1, 3, 4
Suffix-Sharing List S_2 contains 2, 3, 4

S_1 and S_2 share the suffix containing 3 and 4

Prefix-Sharing List P_1 contains 5, 6, 7
Prefix-Sharing List P_2 contains 5, 6, 8

P_1 and P_2 share the prefix containing 5 and 6

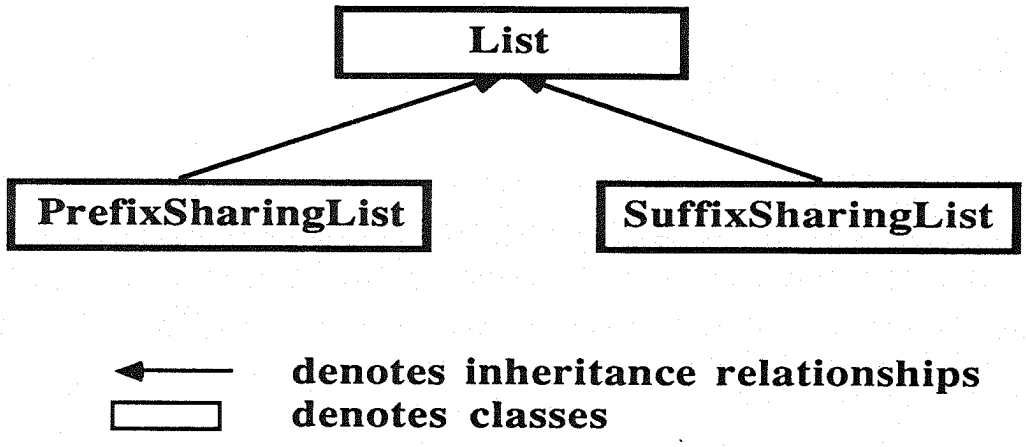
Arrows logically denote sharing links but physically denote pointers

Prefix Versus Suffix-Sharing Lists

Figure 6

Lisp and Prolog currently rely exclusively on suffix-sharing lists. If efficiency is an issue, these lists must be constructed from right-to-left. On the other hand, prefix-sharing lists permit the efficient construction of lists from left-to-right. With appropriate type conversions, the two varieties of lists can be used to support processing directions that match the user's natural inclinations.

As far as the user's perspective is concerned, it makes most sense to devise a logical hierarchy as shown in Figure 7. The existing List class is renamed SuffixSharingList (all references to List are also changed to refer to SuffixSharingList) and a new List class is constructed to play the role of an abstract class which for the moment is devoid of methods. What is noteworthy is that this hierarchy was designed without considering the implementer's perspective (since it hasn't yet been designed).



The List Class Hierarchy (So Far)

Figure 7

The obvious approach to implementing prefix-sharing lists is to produce a mirror-image of the suffix-sharing list implementation. A copy of the suffix-sharing code is made and modified by replacing instance variable names **firstPart** and **restPart** by **lastPart** and **allButLastPart** respectively and also by replacing operation names **first** by **last**, **rest** by **allButLast**, **precede:** by **follow:**, etc. In addition to the name substitutions (a simple editing task), the implementation code for operations that are mirror images; i.e., **complementary** operations, must be reversed; e.g. the code for **first** must be interchanged with the corresponding code for **last**, the code for **rest** by the code for **allButLast**, etc. An operation such as **copy** is its own complementary operation. A sample follows:

Sample Methods For Suffix-Sharing Non-Empty List Exemplar

```
first
    ↑firstPart

last
    self rest isEmpty ifTrue: [↑self first] ifFalse: [↑self rest last]

copy
    ↑self rest copy precede: self first
```

Sample Methods For Prefix-Sharing Non-Empty List Exemplar

```
first
    self allButLast isEmpty ifTrue: [↑self last] ifFalse: [↑self allButLast first]

last
    ↑lastPart

copy
    ↑self allButLast copy follow: self last
```

As a reader, you may have already objected that this approach fails to take inheritance into account. You might have hoped to inherit the suffix-sharing code with perhaps only minor changes to a subset of the operations. The mirror metaphor suggests that keeping the list backwards (as shown in Figure 6) and mapping user requests to the complementary operations is sufficient; i.e., it should be sufficient to map user request **last** to **first, allButLast** to **rest, first** to **last**, etc. as shown below.

Methods For Empty Suffix-Sharing Lists

```
precede: element
    ↑(self class nonEmpty replaceFirst: element) replaceRest: self

follow: element
    ↑(self class empty) precede: element
```

Corresponding Methods For Empty Prefix-Sharing Lists

```
precede: element
    ↑super follow: element

follow: element
    ↑super precede: element
```

The problem is that inheritance introduces double reflections and conspires to prevent this approach from working. For example, if L_1 is an empty prefix-sharing list, "L_1 **precede**: 0" executes as "L_1 **follow**$_S$: 0" (S indicates that it is a suffix-sharing list method) which in turn executes "(L_1 **class empty**) **precede**: element" that maps again to "(L_1 **class empty**) **follow**$_S$: element" since (L_1 **class empty**) is a new empty prefix-sharing list. This leads to an infinite loop. The same problem occurs with "L_1 **follow**: 0" but it occurs when **replaceFirst**: is mapped to **replaceLast**:. In the end, the simplest solution is the first solution indicated above. However, inheritance can used to great advantage at least for empty prefix-sharing lists because the majority of the methods simply signal errors. The only problem is that the error messages are wrong for prefix-sharing lists. This can be repaired by generalizing them. For example, the following method for empty suffix-sharing lists

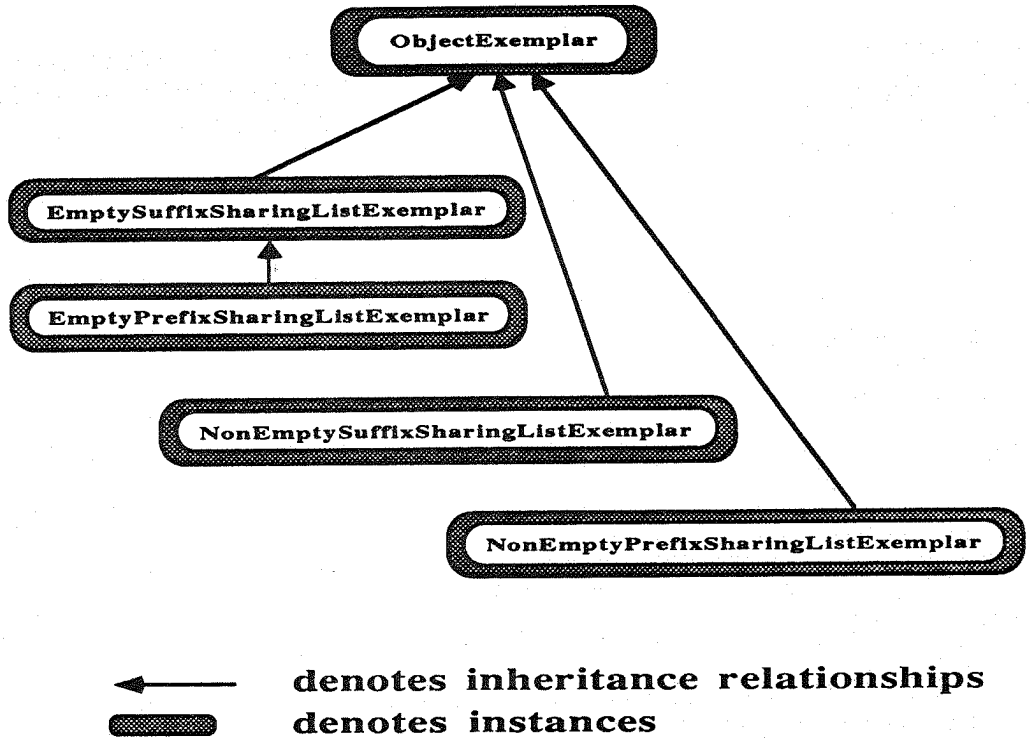
```
first
    ↑self error: 'first is not legal on empty lists'
```

is generalized to

```
first
    ↑self error: 'empty lists do not contain elements'
```

The notion that implementing a new type causes old types to have to be generalized or renamed is unique to systems with inheritance. It is one of the reasons why object-oriented programming is harder than abstract data type designing. The complexity of the generalization task is not the issue; it is the fact that it has to be done at all that makes it noteworthy.

Once done, the implementation of empty prefix-sharing lists is greatly simplified by inheriting from empty suffix-sharing lists. The same advantages do not accrue for non-empty prefix-sharing lists because virtually every method must be re-implemented. Indeed, in order to provide instance variable names like **lastPart** and **allButLastPart**, we must avoid inheriting from non-empty suffix-sharing lists. The hierarchy that results (Figure 8) is asymmetrical, something that is reasonable and acceptable for an implementation hierarchy but likely intolerable had it been a logical hierarchy.



The List Instance Hierarchy (So Far)

Figure 8

In general, this design has one very serious deficiency. Adding a new operation to suffix-sharing lists likely requires the corresponding addition to prefix-sharing lists; i.e., inheritance is not automatic. Such

inter-class dependencies is not something that can be easily designed out. It is another reason why designing families of data types is a programming-in-the-large problem. Better documentation tools is a minimum requirement for handling these issues.

Prefix-Sharing Lists - An Exemplar-Based Implementation

```
class name                                  PrefixSharingList
superclass                                  List
class variable names                        "none"

methods

empty list creation

empty
    ↑EmptyPrefixListExemplar clone

non-empty list creation

nonEmpty
    ↑NonEmptyPrefixListExemplar clone


exemplar name                               EmptyPrefixSharingListExemplar
superexemplar                               EmptySuffixSharingListExemplar
class name                                  PrefixSharingList
instance variable names                     "none"

methods

querying

"Inherit isEmpty, size"

left-side manipulations

"Inherit first, rest"

precede: anObject
    ↑(self class empty) follow: element

prefix: newlength
    | list |
    newlength = 0 ifFalse: [↑self error: 'illegal prefix size'].
    "Must not manufacture a new empty list"
    list ← self.
    [list isEmpty] whileFalse: [list ← list allButLast].
    ↑list

right-side manipulations

"Inherit last, allButLast"
```

```
follow: element
    ↑(self class nonEmpty replaceLast: anObject) replaceAllButLast: self

suffix: newlength
    newlength = 0 ifFalse: [↑self error: 'illegal suffix size'].
    ↑self class empty

other manipulations

"Inherit element:, append:, sublist:for:, sublist:to:, copy"

destructive modifications

"Inherit replaceFirst:, replaceRest:"

"Methods replaceLast:, replaceAllButLast: are left to the reader"

destructiveAppend: list
    | remainingList |
    (list isKindOf: self class) ifFalse: [↑self error: 'cannot append a non-list'].
    "The empty list at the left end of list must become the receiver."
    remainingList ← list.
    [remainingList isEmpty] whileFalse: [remainingList ← remainingList allButLast].
    "Note: the traditional Smalltalk become: is a two-way (swapping) become"
    "X oneWayBecome: Y => Change all and only X-references to Y-references"
    ↑remainingList oneWayBecome: self

sequencing

"Inherit do:, collect:"
```

exemplar name	NonEmptyPrefixSharingListExemplar
superexemplar	ObjectExemplar
class name	PrefixSharingList
instance variable names	lastPart allButLastPart

methods

left to the reader

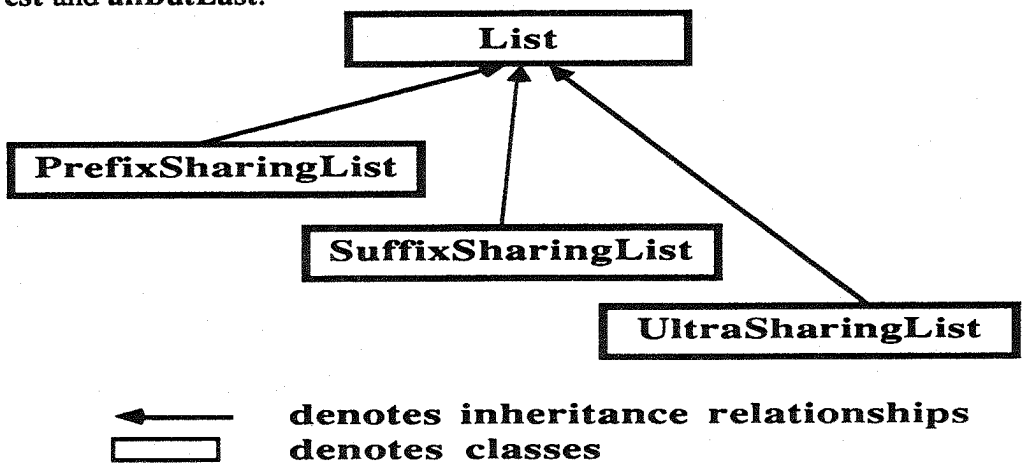
9 Further Extending The Family

The implementation hierarchy need not conform to the same standards of symmetry and neatness as the logical hierarchy.

So far, suffix-sharing lists permit efficient left-to-right traversal and right-to-left construction (right-to-left traversal and left-to-right construction, on the other hand, are very inefficient). Correspondingly, prefix-sharing lists permit efficient right-to-left traversal and left-to-right construction. In both cases, there is a rivalry between traversal and construction directions. Perhaps we could eliminate

the directional bias by concentrating on construction (or traversal) exclusively. If a class of lists can be designed with efficient construction operations without directional bias, the resulting traversal inefficiencies (should a tradeoff be required) might be eliminated as a separate problem.

A quick scan of the operations provides us with operations **precede:**, **follow:**, **prefix:**, **suffix:**, and **append:** as the primary constructors. Each of these needs to construct lists that share in order to be efficient. The solution is to encapsulate the parameters in each case and to encode or "remember" the constructor used. To contrast such lists with the other two varieties of lists, we will call them **ultra-sharing** lists. If we construct an ultra-sharing list with **precede:**, we expect to be able to execute an efficient **rest** on the result but not necessarily an efficient **allButLast**. On the other hand, for an ultra-sharing list constructed with **follow:**, we correspondingly expect to be able to execute an efficient **allButLast** but not a comparably efficient **rest**. We could additionally provide query operations that permit the user to determine how the list was constructed so that more efficient traversal could be done. This would also be useful for efficient use of the destructive operations. Our inclination is to not provide such query operations and to eliminate the destructive operations entirely. Efficient traversal can be provided by recommending (but not insisting) that users use the sequencing operations **do:** and **collect:** instead of **rest** and **allButLast**.



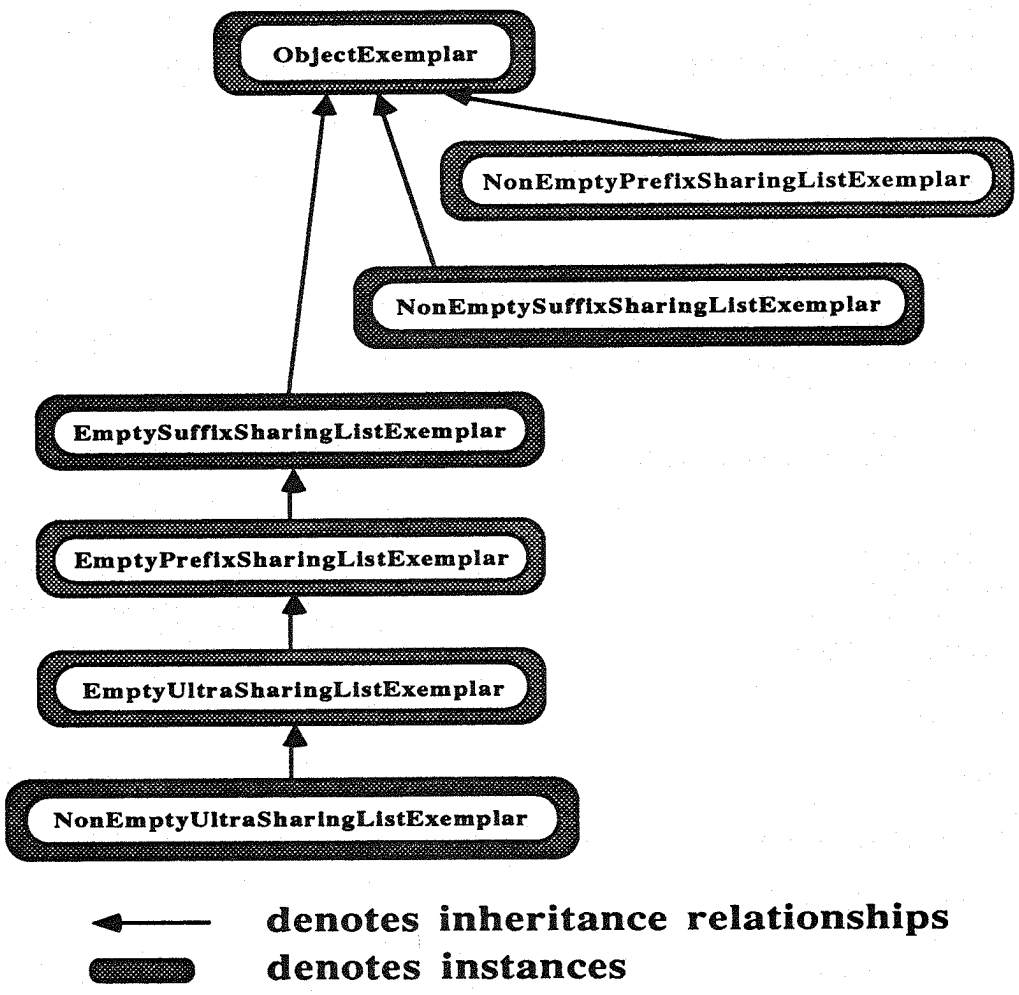
The List Class Hierarchy

Figure 9

The user's perspective is extended by adding class UltraSharingList below List (see Figure 9). However, the implementer's perspective has many more degrees of freedom. Should we inherit from prefix-sharing or suffix-sharing lists? Should the inheritance be asymmetrical like the previous design or not?

If the constructors are analysed in detail, we find that an entirely different design is suggested. The clue comes from the fact that the constructors are the same for both empty and non-empty receivers. For example, **precede:** is implemented identically for both empty and non-empty lists since the receiver (the

list) is simply encapsulated. The most economical solution has non-empty ultra-sharing lists **inheriting the constructors** from empty ultra-sharing lists with the latter inheriting from empty suffix-sharing lists (or empty prefix-sharing lists) as shown in Figure 10.



The List Instance Hierarchy
Figure 10

If the encapsulation strategy is implemented naively, empty ultra-sharing lists could be constructed that are not copies of (and do not inherit from) EmptyUltraSharingListExemplar. For example, appending two empty lists would result in an "encapsulated empty list", an object that inherits from NonEmptyUltraSharingListExemplar. An implementation that accommodates such "encapsulated empty lists" is not difficult but we prefer to eliminate them.

Additionally, most operations now need to interrogate the size of the lists. It seems necessary to explicitly store this size in a **length** field. By contrast, users know that the **size** operation is expensive for prefix-sharing and suffix-sharing list and avoid it; the lack of a **length** field for those data types is both a space saving and a time saving because field updates are not free.

```
class name                                  UltraSharingList
superclass                                  List
class variable names                        "none"

methods

empty list creation

empty
    ↑EmptyUltraSharingListExemplar clone

non-empty list creation

nonEmpty
    ↑NonEmptyUltraSharingListExemplar clone


exemplar name                               EmptyUltraSharingListExemplar
superexemplar                               EmptySuffixSharingListExemplar
class name                                  UltraSharingList
instance variable names                     "none"

methods

querying

"Inherit size, isEmpty"

left-side manipulations

"Inherit first, rest"

precede: element
    | newsize |
    newsize ← self size + 1.
    ↑(self class nonEmpty privateInitialize: self and: #precede and: element and: newsize

prefix: length
    length < 0 | length > self size ifTrue: [↑self error: 'illegal sublist size'].
    length = 0 ifTrue: [↑self class empty].
    ↑(self class nonEmpty privateInitialize: self and: #prefix and: length and: length

right-side manipulations

"Inherit last, allButLast"

follow: element
    | newsize |
    newsize ← self size + 1.
    ↑(self class nonEmpty privateInitialize: self and: #follow and: element and: newsize

suffix: length
```

```
    length < 0 | length > self size ifTrue: [↑self error: 'illegal sublist size'].
    length = 0 ifTrue: [↑self class empty].
    ↑(self class nonEmpty privateInitialize: self and: #suffix and: length and: length
```

other manipulations

"Inherit **element:**, **sublist:for:**, **sublist:to:**, **copy**"

```
append: list
    | newsize |
    (list isKindOf: self class) ifFalse: [↑self error: 'not a valid list for append'].
    newsize ← self size + list size.
    newsize = 0 ifTrue: [↑self class empty].
    ↑(self class nonEmpty privateInitialize: self and: #append and: list and: newsize
```

destructive modifications

```
replaceFirst: element
    ↑self error: 'destructive operations not legal on ultra-sharing lists'

replaceRest: list
    ↑self error: 'destructive operations not legal on ultra-sharing lists'

destructiveAppend: list
    ↑self error: 'destructive operations not legal on ultra-sharing lists'
```

sequencing

"Inherit **do:** and **collect:**"

exemplar name	NonEmptyUltraSharingListExemplar
superexemplar	EmptyUltraSharingListExemplar
class name	UltraSharingList
instance variable names	part1 constructor part2 length
comment	X operation: Y is stored as "part1 constructor part2".

methods

querying

```
isEmpty
    ↑false

size
    ↑length
```

left-side manipulations

```
first
    constructor = #precede ifTrue: [↑part1].
    constructor = #follow ifTrue: [part1 isEmpty ifTrue: [↑part2] ifFalse: [↑part1 first]].
    constructor = #prefix ifTrue: [↑part1 first].
```

```
    constructor = #suffix ifTrue: [↑part1 element: part1 size - part2 + 1].
    constructor = #append ifTrue: [
        part1 isEmpty ifTrue: [↑part2 first] ifFalse: [↑part1 first]

rest
    constructor = #precede ifTrue: [↑part1].
    constructor = #append ifTrue: [part1 size = 1ifTrue: [↑part2]].
    ↑self suffix: self size - 1

"Inherit precede:, prefix:"

right-side manipulations

last
    constructor = #precede ifTrue: [part1 isEmpty ifTrue: [↑part2] ifFalse: [↑part1 last]].
    constructor = #follow ifTrue: [↑part2].
    constructor = #prefix ifTrue: [↑part1 element: part1 size].
    constructor = #suffix ifTrue: [↑part1 last].
    constructor = #append ifTrue: [
        part2 isEmpty ifTrue: [↑part1 last] ifFalse: [↑part2 last]

allButLast
    constructor = #follow ifTrue: [↑part1].
    constructor = #append ifTrue: [part2 size = 1ifTrue: [↑part1]].
    ↑self prefix: self size - 1

"Inherit follow:, suffix:"

other manipulations

element: position
    position < 1 | position > self size ifTrue: [↑self error: 'illegal element position'].
    position = 1 ifTrue: [↑self first].
    position = self size ifTrue: [↑self last].
    constructor = #precede ifTrue: [↑part1 element: position - 1].
    constructor = #follow ifTrue: [↑part1 element: position].
    constructor = #prefix ifTrue: [↑part1 element: position].
    constructor = #suffix ifTrue: [↑part1 element: part1 size - part2 + position].
    constructor = #append ifTrue: [
        part1 size >= position
            ifTrue: [↑part1 element: position]
            ifFalse: [↑part2 element: position - part1 size]]

"Inherit append:, sublist:for:, sublist:to:"

copy
    ↑self collect: [:element | element]

destructive modifications

"Inherit replaceFirst:, replaceRest:, destructiveAppend:"

sequencing
```

```
do: block
    constructor = #precede ifTrue: [block value: part2. part1 do: block].
    constructor = #follow ifTrue: [part1 do: block. block value: part2].
    constructor = #prefix
        ifTrue: [1 to: self size do: [:which | block value: (part1 element: which)]].
    constructor = #suffix
        ifTrue: [
            (part1 size - self size + 1) to: part1 size do: [:which |
                block value: (part1 element: which)]].
    constructor = #append ifTrue: [part1 do: block. part2 do: block]

collect: block
    constructor = #precede ifTrue: [↑(part1 collect: block) precede: (block value: part2)].
    constructor = #follow ifTrue: [↑(part1 collect: block) follow: (block value: part2)].
    constructor = #prefix
        ifTrue: [
            ↑(1 to: self size) inject: (self class empty) into: [:which :result |
                result follow: (block value: (part1 element: which))].
    constructor = #suffix
        ifTrue: [
            ↑(part1 size to: (part1 size - self size + 1) by: -1)
                inject: (self class empty) into: [:which :result |
                    result precede: (block value: (part1 element: which))].
    constructor = #append ifTrue: [↑(part1 collect: block) append: (part2 collect: block)]

private

privateInitialize: aPart1 and: aConstructor and: aPart2 and: aSize
    part1 ← aPart1. constructor ← aConstructor. part2 ← aPart2. length ← aSize.
    ↑self
```

The above design would be more conventional if Smalltalk had the notion of a case statement. We could, of course, eliminate the tests entirely by introducing five representations -- one for each constructor. The ability to trade tests for representations is a new design dimension.

We could further specialize ultra-sharing lists by permitting characters, strings, and ultra-sharing lists built-up from these to be mixed. The result might be called SharableString to distinguish it from existing non-sharable strings. A quick glance back at Table 1 should make it clear that the list operations are indeed typical of strings but not necessarily typical of Smalltalk strings.. The only anomaly is that operation Sublist ought to be called Substring in the string context. Most implementations of strings are non-sharable versions with length information explicitly maintained; e.g., consider PL/1 or nonstandard versions of Pascal [31] that provide strings. Sharable versions such as provided by XPL [20] do exist but they are rare; they are much more efficient than non-sharable implementations but they require a garbage collected system, a primary component of object-oriented systems.

We won't actually pursue this specialization here because additional work would have to be done to unify the new sharable string with the old non-sharable string design; e.g., two semantically equivalent operations might have different names, some operations specific to character strings might usefully be

generalized to lists, etc.

The three varieties of lists completed so far along with non-sharable strings is not intended to be a complete inventory of possible designs. Special applications are likely to come up with additional varieties that have even more specific space-time tradoffs. Our intent was to expose the reader to the many options available in an exemplar-based system and to make it quite clear that the implementation hierarchy is divorced from the logical hierarchy.

10 On The Need For Familial Classes

By providing direct access to its many subclasses, familial classes help to decrease the name space size of the class working set.

In the context of a large evolving class-based system, the only way to discover and understand the system is to browse through the class hierarchy. Even with a sophisticated tool, the hierarchy can be bewildering through sheer numerical size. Part of the difficulty is that the important information is distributed; there is no central authority that compares and contrasts the members of important families. From the point of view of users, there is no recourse but to browse all classes. Numerically, the number of families is smaller than the number of classes by a substantial factor; e.g., the list family may have 3 or 4 specializations, the array family 5 or 6, the stack family 2 or 3, the set family 2 or 3, the window family ... The fundamental problem is that the *name space is too large for typical users.*

To begin to address this problem, we can institute **familial classes**; i.e., classes whose primary role is to document, interface with, and mediate between the many members of the family. If members themselves are familial classes, a proper sharing of duties can be specially decided upon. The familial class not only provides the default behaviour for the more popular specialization but also provides for alternative ways of communicating with the subclasses. In the extreme, all communication between users and the different varieties could be directed through the familial class. For example, if a particular kind of empty list is desired, rather than communicate directly with the specific variety of lists, users could communicate with the familial class List.

The notion of List as a familial class is quite different from the notion of List as an abstract class. For example, Collection in Smalltalk is currently an abstract class. However, it plays no role as an intermediary between users and Collection specializations. If a user asks for an empty collection, he does not get a useful default such as an OrderedCollection. Instead, he gets an error message indicating that it is an abstract class and consequently does not instantiate new instances. In fact, there is no useful communication that can ensue between users and the abstract Collection class. This is because its role is purely to simplify the implementation; i.e., its only purpose is to serves as a repository for methods that can be usefully shared between all specializations. In an examplar-based system, abstract instances play

the same role. Although they could be used as a repository for class methods, corresponding abstract classes are rarely needed because of the relative paucity of class methods.

By decreeing that a specific class be a familial class, we are insisting that special methods be provided in the class for communicating with the members of the family. Although we don't have any specific insights as to how these methods ought to look like generally, we do emphasize that the language of selection need not and probably should 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) from List in any one of the following ways:

- List **empty**
- List **standardEmpty**

- List **suffixSharingEmpty**
- List **leftToRightTraversableEmpty**
- List **rightToLeftConstructableEmpty**

- List **prefixSharingEmpty**
- List **rightToLeftTraversableEmpty**
- List **leftToRightConstructableEmpty**

- List **ultraSharingEmpty**
- List **biTraversableEmpty**
- List **biConstructableEmpty**

By choosing suffix-sharing lists to be the default, normal usage will avoid the more specialized names. Only special applications will make use of the others. With complex families, 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 implementation of the varieties is also a worthwhile goal. Better use of names that combine attributes like **fast, small, fastAndBounded, fastAndUnbounded,** ... is a notion that the programming community has little experience with so far.

The Collection family in Smalltalk is distinguished because it contains several subfamilies. We can envisage Collection, Array, and Set being familial classes. Classes such as Symbol and String, because of their special role, might be accessible via Array but could also be viewed as degenerate familial classes; i.e., conventional classes denoting only one member. On the other hand, introducing sharable strings in addition to the non-sharable variety would likely promote string to a more "familial" role.

In fact, familial classes provide for the notion of multiple perspectives; i.e., sharable strings could be accessed either from the perspective of a special kind of lists from the List familial class or from the perspective of a string variation from the String familial class (where the default is the standard non-sharable string).

11 Designing Families Without Class Specializations.

The separation of logical hierarchies from implementation hierarchies along with the notion of familial classes permits fine control over the class name space.

With the notion of familial classes, it should be clear that SuffixSharingList, PrefixSharingList, and UltraSharingList can actually be eliminated from the class name space. To remove them from the class hierarchy, three modifications are required:

- Methods **emptyClone** and **nonEmptyClone** are added to each instance exemplar; e.g., for suffix-sharing lists, the first is implemented to return "EmptySuffixSharingListExemplar **clone**" and the second "NonEmptySuffixSharingListExemplar **clone**".
- Corresponding code of the form "self **class empty**" and "self **class nonEmpty**" is replaced by "self **emptyClone**" and "self **nonEmptyClone**" respectively.
- Class based tests are replaced by tests based on instance exemplars; e.g., "X **isMemberOf**: SuffixSharingList" is replaced by "(X **isCloneOf**: EmptySuffixSharingListExemplar) **or**: [X **isCloneOf**: NonEmptySuffixSharingListExemplar]".

Without specializations in the logical hierarchy, there may be little point in calling List a familial class; i.e., List is equally well considered a normal class with a large number of alternative representations for the instances, each designed to satisfy different space/time tradeoffs.

Removing all classes is the opposite extreme to having a class for each different instance. In fact, the previous design was a compromise between the two extremes. Only exemplar-based systems permit this kind of flexibility.

12 Conclusions

An exemplar-based Smalltalk already provides effective support for programming-in-the-large but additional mechanisms and features are needed to provide enhanced capabilities.

Designing families of data types requires attention to details and the use of facilities that programming-in-the-small need not concern itself with:

- *Ensuring that the data type name space remain small.* As networks of users begin to exchange and develop software, the name space is bound to grow beyond expectations. It is essential to take this into account before the fact rather than after.
- *Designing familial classes that capture the essence of entire families.* If familial classes serve as ports to large numbers of classes, they will serve to centralize information about

these classes. Documenting their commonalities and differences from the user's point of view will serve to make users more knowledgeable about the system. The system itself will also appear simpler.

- *Weighing tradeoffs between implementations with multiple classes and alternatives with multiple representations.* This design dimension is new to exemplar-based systems.

- *Distinguishing between logical hierarchies and implementation hierarchies.* Class hierarchies should describe logical relationships without revealing the implementation. Implementation hierarchies, since they are decoupled, are free to pursue strategies unrelated to the logical view -- a new degree of freedom that class-based systems do not provide.

- *Designing strongly coupled implementation hierarchies.* Designing efficient and effective implementation hierarchies requires deep knowledge about many related exemplars. Programming by exemplars is relatively new. The current design methodology could be described as designing with the one-exemplar paradigm. The multiple-exemplar paradigm provides a new direction.

Our aim was to bring into focus some of the problems and issues that should be addressed when designing data types for integration into families. We have argued that class-based systems provide inadequate facilities for both implementers and users and we have highlighted their major deficiencies. We have also shown how exemplar-based systems remedy the deficiencies and discussed the additional degrees of design freedom they provide. Our discussion should make it evident that switching from class-based systems to exemplar-based systems is one way of providing more powerful object-oriented capabilities. Designers of new object-oriented systems should carefully investigate whether or not they wish to provide a traditional class-based system or a more powerful exemplar-based system. Actually converting class-based systems to exemplar-based systems is not a major task.

Acknowledgements

The research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada), DREA (Defense Research Establishment at Atlantic), and DREO (Defense Research Establishment at Ottawa). We would also like to thank the members of the Actra project who contributed to the evolution of these ideas: Dave Thomas, John Pugh, Mike Wilson, John Duimovich, Peter Shipton, and Jeff McAffer.

References

1. 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 1981).
2. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M.J., and Zdybel, F. CommonLoops: Merging Common Lisp and object-oriented programming. Xerox Palo Alto Reseatch Center: Intelligent Systems Laboratory Series ISL-85-8 (August 1985).
3. Borning, A. *The programming language aspects of Thinglab, a constraint-oriented simulation laboratory.* ACM Toplas, Vol. 3, No. 4 (Oct. 1981), 353-387.
4. 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 1982), 21-23.
5. Cannon, H.I. Flavors. Technical Report, MIT Artificial Intelligence Lab. (1980).
6. Cox, B. Message/object programming: An evolutionary change in programming technology. IEEE Software, Vol. 1, No. 1 (Jan 84), 50-61.
7. DeRemer, F. and Kron, H. Programming-in-the-large versus programming-in-the-small. IEEE Transactions on Software Engineering, SE-2 (June 1976), 80-86.
8. Duff. C. Neon - Extending Forth in new directions. Proc. of 1984 Asilomar FORML Conf. (1984).
9. Goldberg, A. and Robson, D. *Smalltalk-80: The language and its implementation.* Addison-Wesley, Reading, Mass. (1983).
10. Goldberg, A. *Smalltalk-80: The interactive programming environment.* Addison-Wesley, Reading, Mass. (1984).
11. Green, M. and Philp, P. The use of object-oriented languages in graphics programming. Proceedings of NCGA Graphics Interface 1982 Conference, Toronto (1982).
12. Klahr, P., McArthur, D. and Narian, S. SWIRL: An object-oriented air battle simulator. Proc. AAAI-82, Pittsburgh (August 1982).
13. Krasner, G. *Smalltalk-80: Bits of history, words of advice.* Addison-Wesley, Reading, Mass. (1983).
14. 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).
15. LaLonde, W.R.*Why examplars are better than classes*. Technical Report SCS-TR-93, School of Computer Science, Carleton University (May 1986).
16. LaLonde, W.R., Thomas, D.A., and Pugh, J.R. *An exemplar based Smalltalk.* First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon (September 1986).
17. Lieberman, H. A preview of Act 1. MIT AI Laboratory Memo No. 625 (June 1981).
18. Lieberman, H. Thinking about lots of things at once without getting confused - parallelism in Act 1. MIT AI Laboratory Memo No. 626 (May 1981).
19. Lieberman, H. *Using prototypical objects to implement shared behavior in object oriented systems.* First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon (September 1986).
20. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A compiler generator.* Prentice-Hall (1970).
21. O'Brien, P. Trellis: Object-based environment, Language Tutorial. Eastern Research Lab., Digital Research Lab Tech. Rep. DEC-TR-373 (Nov. 1985).
22. Pugh, J.R. and LaLonde, W.R. Data structures and data types: an object-oriented approach. CIPS Congress '86, Vancouver, B.C. (April 1986), 251-258.
23. Reynolds, C.W. Computer animation with scripts and actors. Proceeding of ACM SIGGRAPH Conference (July 1982).
24. Shapiro, E.Y. and Takeuchi, A. Object-oriented programming in Concurrent Prolog. New Generation Computing, OHMSHA Ltd and Springer-Verlag, Vol. 1 (1983), 25-48.
25. Smith, B. Reflections and semantics in a procedural language. M.I.T. Laboratory for Computer Science Report MIT-TR-272, (1982).
26. Stefik, M.J. and Bobrow, D.G. Object-oriented programming: Themes and variations. AI Magazine, Vol. 6, No. 4 (1986), 40-62.
27. Stroustrup, B. The C++ reference manual. Addison-Wesley, 1986.
28. Tesler, L. Object-Pascal report. Apple Computer (Feb. 1984).
29. Thalmann, D. and Magnenat-Thalmann, N. Actor and camera data types in computer animation. Proc. Graphics Interface 83, Edmonton, Canada (1983).
30. Thomas, D.A. and Lalonde, W.R. *Actra: The design of an industrial fifth generation Smalltalk system.* Proc. of IEEE COMPINT '85, Montreal, Canada (Sept. 1985), 138-140.
31. *Turbo Pascal,* Version 3, Reference Manual. Borland International Inc. (1985).
32. Vaucher, J.G. and Lapalme, G. *POOPS: Object oriented programming in Prolog.* Technical Report 565, Laboratoire INCOGNITO, Dept. d'Informatique et de Recherche Operationnelle, University of Montreal (March 1986).
33. Weinreb, D. and Moon, D. Flavours - message-passing in the Lisp Machine. MIT AI Memo No. 602 (Nov. 1980).
34. Zaniolo, C. Object-oriented programming in Prolog. 1984 International Symposium on Logic Programming, New Jersey (Feb. 1984), 265-271.

Carleton University, School of Computer Science
Bibliography of Technical Reports
Publications List (1985 –>)

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
KIS 5B6

SCS-TR-66 **On the Futility of Arbitrarily Increasing Memory Capabilities of Stochastic Learning Automata**
B.J. Oommen, October 1984. Revised May 1985.

SCS-TR-67 **Heaps in Heaps**
T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 **Partial Orders and Comparison Problems**
out-of-print M.D. Atkinson, November 1984. See Congressus Numerantium 47 ('86), 77-88

SCS-TR-69 **On the Expected Communication Complexity of Distributed Selection**
N. Santoro, J.B. Sidney, S.J. Sidney, February 1985.

SCS-TR-70 **Features of Fifth Generation Languages: A Panoramic View**
Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 **Actra: The Design of an Industrial Fifth Generation Smalltalk System**
David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 **Minmaxheaps, Orderstatisticstrees and their Application to the Coursemarks Problem**
M.D. Atkinson, J.-R. Sack, N. Santoro, T. Strothotte, March 1985.

SCS-TR-73 **Designing Communities of Data Types**
Wilf R. LaLonde, May 1985.
Replaced by SCS-TR-108

SCS-TR-74 **Absorbing and Ergodic Discretized Two Action Learning Automata**
out-of-print B. John Oommen, May 1985. See IEEE Trans. on Systems, Man and Cybernetics, March/April 1986, pp. 282-293.

SCS-TR-75 **Optimal Parallel Merging Without Memory Conflicts**
Selim Akl and Nicola Santoro, May 1985

SCS-TR-76 **List Organizing Strategies Using Stochastic Move-to-Front and Stochastic Move-to-Rear Operations**
B. John Oommen, May 1985.

SCS-TR-77 **Linearizing the Directory Growth in Order Preserving Extendible Hashing**
E.J. Otoo, July 1985.

SCS-TR-78 **Improving Semijoin Evaluation in Distributed Query Processing**
E.J. Otoo, N. Santoro, D. Rotem, July 1985.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-79 **On the Problem of Translating an Elliptic Object Through a Workspace of Elliptic Obstacles**
B.J. Oommen, I. Reichstein, July 1985.

SCS-TR-80 **Smalltalk - Discovering the System**
W. LaLonde, J. Pugh, D. Thomas, October 1985.

SCS-TR-81 **A Learning Automation Solution to the Stochastic Minimum Spanning Circle Problem**
B.J. Oommen, October 1985.

SCS-TR-82 **Separability of Sets of Polygons**
Frank Dehne, Jörg-R. Sack, October 1985.

SCS-TR-83 **Extensions of Partial Orders of Bounded Width**
out-of-print M.D. Atkinson and H.W. Chang, November 1985. See Congressus Numerantium, Vol. 52 (May 1986), pp. 21-35.

SCS-TR-84 **Deterministic Learning Automata Solutions to the Object Partitioning Problem**
B. John Oommen, D.C.Y. Ma, November 1985

SCS-TR-85 **Selecting Subsets of the Correct Density**
out-of-print M.D. Atkinson, December 1985. To appear in Congressus Numerantium, Proceedings of the 1986 South-Eastern conference on Graph theory, combinatorics and Computing.

SCS-TR-86 **Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacles Case**
B. J. Oommen, S.S. Iyengar, S.V.N. Rao, R.L. Kashyap, February 1986

SCS-TR-87 **Breaking Symmetry in Synchronous Networks**
Greg N. Frederickson, Nicola Santoro, April 1986

SCS-TR-88 **Data Structures and Data Types: An Object-Oriented Approach**
John R. Pugh, Wilf R. LaLonde and David A. Thomas, April 1986

SCS-TR-89 **Ergodic Learning Automata Capable of Incorporating Apriori Information**
B. J. Oommen, May 1986

SCS-TR-90 **Iterative Decomposition of Digital Systems and its Applications**
Vaclav Dvorak, May 1986.

SCS-TR-91 **Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-92 **ACTRA - A Multitasking/Multiprocessing Smalltalk**
David A. Thomas, Wilf R. LaLonde, and John R. Pugh, May 1986

SCS-TR-93 **Why Exemplars are Better Than Classes**
Wilf R. LaLonde, May 1986

SCS-TR-94 **An Exemplar Based Smalltalk**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-95 **Recognition of Noisy Subsequences Using Constrained Edit Distances**
B. John Oommen, June 1986

SCS-TR-96 **Guessing Games and Distributed Computations in Synchronous Networks**

J. van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, June 1986.

SCS-TR-97 **Bit vs. Time Tradeoffs for Distributed Elections in Synchronous Rings**
M. Overmars and N. Santoro, June 1986.

SCS-TR-98 **Reduction Techniques for Distributed Selection**
N. Santoro and E. Suen, June 1986.

SCS-TR-99 **A Note on Lower Bounds for Min-Max Heaps**
A. Hasham and J.-R. Sack, June 1986.

SCS-TR-100 **Sums of Lexicographically Ordered Sets**
M.D. Atkinson, A. Negro, and N. Santoro, May 1987.

SCS-TR-102 **Computing on a Systolic Screen: Hulls, Contours, and Applications**
F. Dehne, J.-R. Sack and N. Santoro, October 1986.

SCS-TR-103 **Stochastic Automata Solutions to the Object Partitioning Problem**
B.J. Oommen and D.C.Y. Ma, November 1986.

SCS-TR-104 **Parallel Computational Geometry and Clustering Methods**
F. Dehne, December 1986.

SCS-TR-105 **On Adding *Constraint Accumulation* to Prolog**
Wilf R. LaLonde, January 1987.

SCS-TR-107 **On the Problem of Multiple Mobile Robots Cluttering a Workspace**
B. J. Oommen and I. Reichstein, January 1987.

SCS-TR-108 **Designing Families of Data Types Using Exemplars**
Wilf R. LaLonde, February 1987.

SCS-TR-109 **From Rings to Complete Graphs - Θ(n logn) to Θ(n) Distributed Leader Election**
Hagit Attiya, Nicola Santoro and Shmuel Zaks, March 1987.

SCS-TR-110 **A Transputer Based Adaptable Pipeline**
Anirban Basu, March 1987.

SCS-TR-111 **Impact of Prediction Accuracy on the Performance of a Pipeline Computer**
Anirban Basu, March 1987.

SCS-TR-112 **ε-Optimal Discretized Linear Reward-Penalty Learning Automata**
B.J. Oommen and J.P.R. Christensen, May 1987.

SCS-TR-113 **Angle Orders, Regular n-gon Orders and the Crossing Number of a Partial Order**
N. Santoro and J. Urrutia, June 1987.

SCS-TR-115 **Time Is Not a Healer: Impossibility of Distributed Agreement in Synchronous Systems with Random Omissions**
N. Santoro, June 1987.

SCS-TR-116 **A Practical Algorithm for Boolean Matrix Multiplication**
M.D. Atkinson and N. Santoro, June 1987.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-117 **Recognizing Polygons, or How to Spy**
James A. Dean, Andrzej Lingas and Jörg-R. Sack, August 1987.

SCS-TR-118 **Stochastic Rendezvous Network Performance - Fast, First-Order Approximations**
J.E. Neilson, C.M. Woodside, J.W. Miernik, D.C. Petriu, August 1987.

SCS-TR-120 **Searching on Alphanumeric Keys Using Local Balanced Trie Hashing**
E.J. Otoo, August 1987.