An Exemplar Based Smalltalk

Wilf R. LaLonde, Dave A. Thomas and John R. Pugh
School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract Two varieties of object-oriented systems exist: one based on classes as in Smalltalk and another based on exemplars (or prototypical objects) as in Act/1. By converting Smalltalk from a class based orientation to an exemplar base, independent instance hierarchies and class hierarchies can be provided. Decoupling the two hierarchies in this way enables the user's (logical) view of a data type to be separated from the implementer's (physical) view. It permits the instances of a class to have a representation totally different from the instances of a superclass. Additionally, it permits the notion of multiple representations to be provided without the need to introduce specialized classes for each representation. In the context of multiple inheritance, it leads to a novel view of inheritance (or-inheritance) that differentiates it from the more traditional multiple inheritance notions (and-inheritance). In general, we show that exemplar based systems are more powerful than class based systems. We also describe how an existing class based Smalltalk can be transformed into an exemplar-based Smalltalk and discuss possible approaches for the implementation of both and-inheritance and or-inheritance.

1 Introduction

Although Smalltalk [Goldberg 83] is a small, well designed language with a rich programming environment, it lacks some of the generality that might be expected of an object-oriented system. In particular, it fails to distinguish between class hierarchies and instance hierarchies. For a great majority of the classes, the distinction is not important; however, there are some important anomalies and consequences. An exemplar based system [LaLonde 86] can distinguish between the two and properly handle these anomalies. It can also be used to support the "classical" class viewpoint.

This research has been supported by DREA, DREO, and NSERC.

In this paper, we detail some of the Smalltalk deficiencies that are a direct result of its class based orientation and describe how a more general and flexible system is obtained by changing its underlying organizational base from classes to exemplars. The work is relevant to the evolution of any object-oriented system including the Flavour system in Lisp [Weinreb 80], Loops [Bobrow 81], Trellis [O'Brien 85], and more conventional languages being retro-fitted with object-oriented facilities; e.g., C [Cox 84], Pascal [Tesler 84], Forth [Duff 84], and Prolog [Shapiro 83, Vaucher 86]. Exemplar based systems include Act/1[Lieberman 81, Smallworld [Laff 81], Thinglab [Borning 82], and Actra [Thomas 85].

This work is part of a research and development project concerned with the design and implementation of Actra [Thomas 85], an object-oriented multiprocessor system based on a version of Smalltalk that supports exemplars and actors (concurrently executable objects). Actra is targetted for use in industrial applications such as flexible manufacturing, simulation and training, command and control, CAD/CAM and project management. An existing uniprocessor version of Actra supporting actors is currently being migrated to a multiprocessor and converted from a class based system to an exemplar based system.

2 Classes Versus Exemplars

Intuitively, a **class** is a data type positioned within an inheritance hierarchy. The hierarchy describes both the relationship between the class and other classes (the inheritance of class variables and class methods) and the relationhip between its instances and other instances (the inheritance of instance variables and instance methods). It can be described by

1. an object denoting the superclass,
2. a set of class variables and pool variables (public to instances and classes),
3. a set of class instance variables (private class representation),
4. a set of class methods (class operations),
5. a set of instance variables (instance representation),
6. a set of instance methods (instance operations).

Class and pool variables are extra to the class notion. Making variables public is a function of the compiler symbol

table lookup routine, not the basic class mechanism. Classes simultaneously describe two hierarchies:

1. A **class hierarchy** that relates the different classes. Each class inherits all class variables and class methods of the classes higher up in the hierarchy; i.e., from the superclass, its superclass, ...

2. An **instance hierarchy** that relates the different instances of the classes in the class hierarchy. Each instance inherits all instance variables and instance methods of the instances associated with the classes higher up in the hierarchy.

The two hierarchies are clearly intertwined in a one-to-one correspondence. As we will see, it is the strong coupling between these two hierarchies that diminishes the flexibility of the system.

Intuitively, an **exemplar** (or **prototype**) [Borning 81] is an example instance (also a sample or prototypical instance) which can serve as a role model for other instances. It is completely described by

1. an object denoting its superexemplar,
2. an object denoting its class,
3. a set of variables (the representation),
4. a set of methods (the operations).

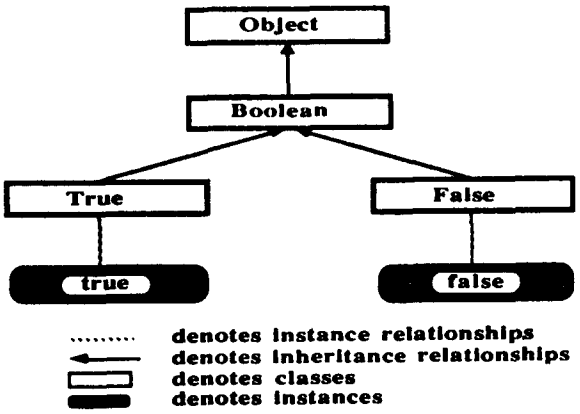
Inheritance (both for variables and methods) proceeds along the superexemplar chain, not along the class hierarchy. The class is maintained to provide access to classification information, class variables and other exemplars. To define a new data type in a hierarchy, we must create two exemplars: a **class exemplar** and an **instance exemplar**. Typically, the class exemplar will inherit from another class exemplar and the instance exemplar from an instance exemplar. Thus, the two hierarchies are distinct.

In the exemplar model, a class can have any number of exemplars associated with it, each with a different representation but typically with the same methods (although implemented differently to reflect the different representation). One of these is nominated the **standard** exemplar. Instances are obtained by **cloning** exemplars (typically, only instance exemplars are cloned). Classes in the exemplar model can also provide new instances; however, this is achieved by having the class clone its standard exemplar. Conventional Smalltalk classes can be viewed as classes with exactly one examplar. The notion of multiple representations can be accommodated with different exemplars instead of with different subclasses. A case for multiple representations might be a bit map which can either be in a standard representation, a compact representation (run-length encoded), or a disk resident representation.

For a simple but more detailed example, consider the definition of a class Boolean with two instances **true** and **false**. Figures 1 presents a class based design while Figure 2 presents an exemplar based approach. For efficiency reasons, each instance is given a different implementation of the methods. For illustration, the **and:** method for each instance is shown below:

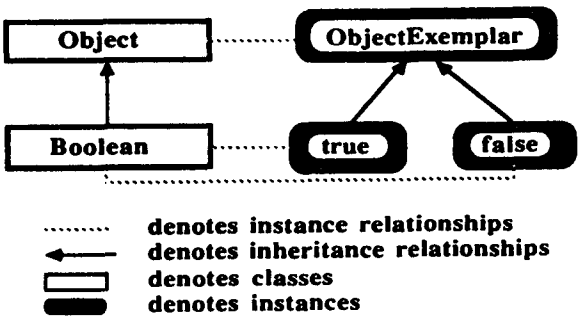
"for **true**"	"for **false**"
and: anotherBoolean	**and:** anotherBoolean
^anotherBoolean	^**false**

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



A Class Based Design

Figure 1



An Exemplar Based Design

Figure 2

False and instance **false**). Although not particularly useful, it would be possible to permit multiple instances of **true** and **false**.

In the 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 (a typical object).

3 Why The Class Based Model Is Inadequate?

Because the exemplar based model permits the class hierarchy to be decoupled from the instance hierarchy, it is clearly less restrictive. This in itself is not sufficient to indict the class based models because this additional flexibility may not be needed. Lacking such flexibility, however, has some obvious implications (in increasing order of importance).

1. All instances of a specific class must have identical representations and methods. Thus, instances cannot have specialized methods and multiple representations for instances are not possible.

2. Specializations of classes with individualized representations are not allowed. Subclasses must have a representation that includes the superclass representation.

3. Since the class hierarchy and instance hierarchy are intertwined by design, either the class hierarchy must be made to conform to the instance hierarchy or the instance hierarchy made to conform with the class hierarchy. Either can involve unacceptable tradeoffs.

The multiple representations problem (case 1) can be resolved by introducing distinct classes for each representation. The simple solution (where possible) is to make each new class a subclass of the original as was done in the Boolean example. Because of case 2, this is not always possible. To avoid inheriting the representation of the original class, it may be necessary to make the new class a subclass of Object (or some other logically unrelated class). If we do, this is an example of case 3 where the logical hierarchy was made to conform with the instance hierarchy. Because of the importance of this last case, we will discuss it in more detail with illustrated examples.

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.

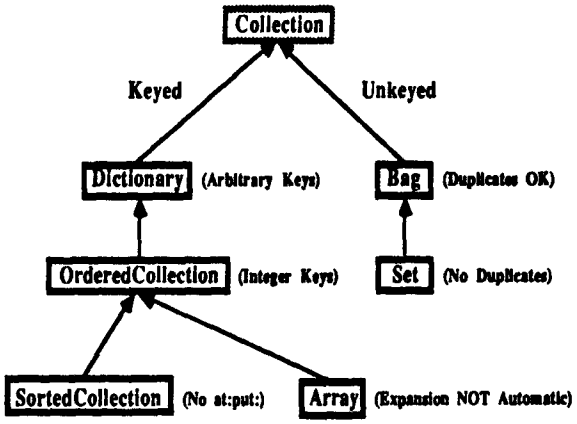
2. It may have a special-purpose representation for more efficient storage and/or access.

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. However, as designers, they must also attempt to realize the intended behaviour as efficiently as possible. Generally, this means sharing as much as possible from existing classes, using the most efficient representations and algorithmns, and perhaps even resorting to a trick or two to achieve the goal. The implementers' concerns are quite different from the users' concerns. Problems arise as soon as the two concerns begin to conflict.

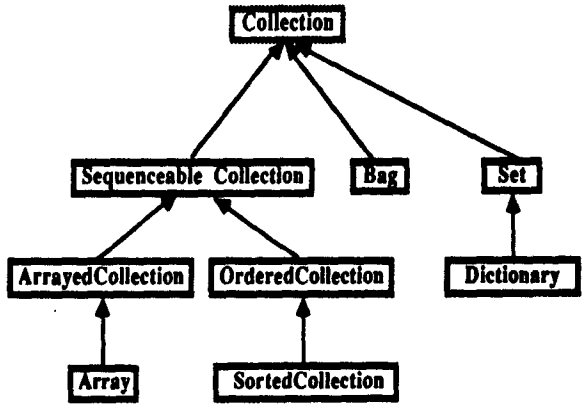
For illustration, consider a subset of the Collection classes available in the Smalltalk system. Figure 3 illustrates a user's view of these classes whereas Figure 4 presents an implementer's view. Before we begin to analyse the classes, we would like to emphasize that the resulting organization is a consequence of the class base, not a consequence of poor design choices.

First, consider classes Set and Bag. Logically, Set is a specialization of Bag (a bag is a set that allows duplicates). Both should have the same set-like operations such as union, intersection, difference. However, they must have slightly different semantics for the insert operation. Bags, for example, insert all submissions. Sets, on the other hand, permit the insertion only if no duplicate exists; i.e., inserting a duplicate has no effect. A straightforward implementation for Set would



A Subset of the Classes: A Logical View

Figure 3



A Subset of the Classes: A Physical View

Figure 4

simply inherit the representation for Bag; e.g., if Bag maintained counts for the number of duplicates, this count could be restricted to 1 for Sets. A better representation, 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 an understandable 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. 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 (its operations and their semantics) and its implementation (its representation and coding). A similar notion is needed for object-oriented systems. Classes should be organized in a manner that reflects the logical relationships of the members. This relationship should not change every time an implementation is modified.

4 Why Exemplars Are Desirable?

It should be clear that exemplars can easily be used to solve each of the above problems. The logical relationships are maintained via class exemplars and the implementation strategies are realized via instance exemplars.

For example, class Set can be made to inherit from class Bag to establish the logical relationship between the two. 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 same approach could be used for Arrays and Dictionaries.

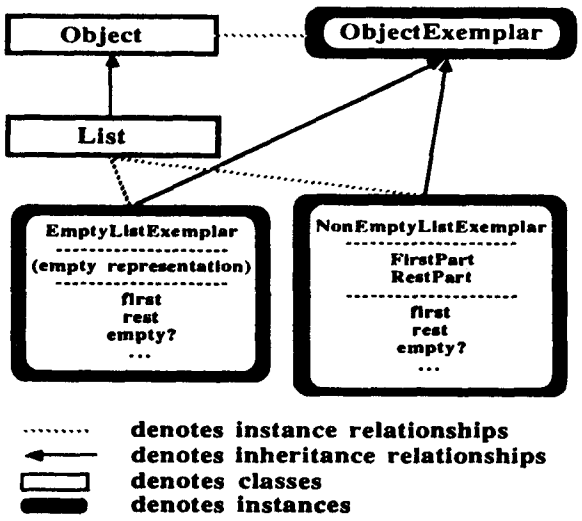
The exemplar base is superior primarily because it permits a separation between users and implementers. In the short term, such separation may not matter but in the long term, it will be crucial. As the applications and additions begin to accumulate in public libraries, users will have neither the time, the patience, or the ability to determine how to use a new family of data types by navigating the implementation. More likely, useful help systems will be designed that help document the essential features and operations of important classes. When the time comes, users will find it much easier to understand and remember relationships that are logical rather than physical; i.e., side effects of specific implementation design decisions.

Unlike the class base, the exemplar base provides a more malleable environment for changes. Changes to the representation of instance exemplars can be accommodated without impacting the class hierarchy. Conversely, the class structure can be modified without impacting the instances. Evolutionary changes are handled much better by the more flexible exemplar base.

5 Exemplars Support Multiple Representations

To illustrate the multiple representation issue, consider the definition of Lisp-style lists with two distinct representations: one for empty lists and another for non-empty lists (see Figure 5).

The List class could be defined by cloning an existing class exemplar. The empty list exemplar is defined with an empty representation (no instance variables) and all the usual methods such as **first** and **rest** (both reporting error), **empty?** (returning **true**), etc. Similarly, the non-empty list exemplar is defined with a two component representation (a first part and a rest part) along with the same methods as above. In this case, however, **first** would return the first part, **rest** the rest part, **empty?** would return **false**, etc. Both instance exemplars would be initialized with the List class as their denoted class. They would also be added to the List class as a member of the set of instance exemplars with the empty list exemplar designated the standard exemplar. Additional methods might also be added to the List class to make it more complete; e.g., method **empty** which returns a clone of the empty list exemplar.



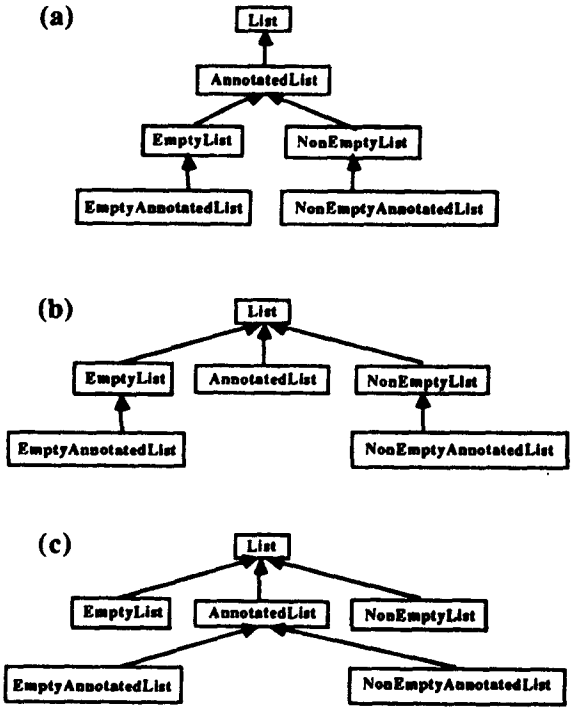
Lists With Multiple Representations

Figure 5

There are advantages to using several exemplars instead of the traditional one. First, their use can play a significant role in speed optimizations; e.g., in the list example above, instances no longer need to perform run-time conditional checks to distinguish between empty lists and non-empty lists. Second, they provide a realization of multiple representations; e.g., by permitting packed and unpacked representations without adding to the already large class name space, by permitting separate memory-based and disk-based representations -- options that could play an important role as object-oriented databases are developed. The ability to provide multiple representations is of particular importance in industrial applications such as CAD/CAM or flexible manufacturing.

An argument that is often presented is that the above example can just as easily be constructed with the class base 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 are designed. For example, suppose we wish to introduce a new class, AnnotatedList say, as a specialization of List (it has additional special purpose instance variables) with its two corresponding instance exemplars: EmptyAnnotatedList and NonEmptyAnnotatedList.

We expect NonEmptyAnnotatedList to be below NonEmptyList (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

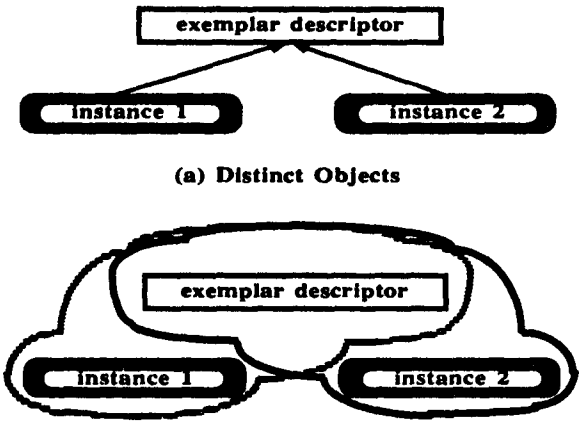
Figure 6

Several designs are possible (see Figure 6). Each has its own deficiencies. Placing AnnotatedList between List and its two specializations (Figure 6a) causes standard lists to be erroneously viewed as annotated lists. Placing it below List and beside its two specializations (Figure 6b) has the implication that EmptyAnnotatedList, for example, is not an AnnotatedList. The only remaining solution is to place AnnotatedList below List (Figure 6c) with EmptyAnnotatedList and NonEmptyAnnotatedList below AnnotatedList. However, this is unacceptable because virtually all standard list methods must be duplicated. The only acceptable solutions either use 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 becomes increasingly problematical. The issue does not arise in exemplar based systems sinces classes and instances form separate hierarchies.

6 Representation Of Exemplars

Two related but fundamentally different techniques for representing exemplars are possible (see Figure 7): one partitions exemplars into two physically distinct components each capable of being manipulated as objects; the other encapsulates the two components into one using a sharing mechanism.



Representation of Exemplars

Figure 7

In the first approach, rather than maintain all relevant information about exemplars in a class, an **exemplar descriptor** is used to fulfill the equivalent role. The representation of objects is left unchanged to minimize modifications to the existing system but the class object pointer is replaced by a pointer to the exemplar descriptor. The class associated with an exemplar and the superexemplar is maintained in the descriptor along with the owner of the descriptor. An instance is an exemplar if and only if it is the owner of its respective exemplar descriptor. Cloning an instance involves copying the instance but not the exemplar descriptor. Creating a new exemplar involves creating both a new instance and a new exemplar descriptor.

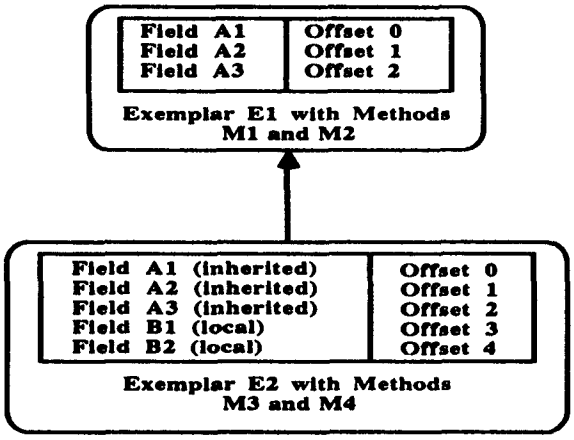
In the second approach, exemplar descriptors are not permitted to be individual objects. Instead, exemplars are objects with a distributed representation. All instances cloned from the same exemplar are viewed as containing and sharing the same exemplar descriptor information; i.e., those fields that used to be part of the exemplar descriptor in the first approach are now part of each instance but each field exists only once for the exemplar and its clones. In more general exemplar based systems such as described in [Lieberman 86], one could envisage providing support for user definable distributed objects with shared components. Although attractive in its own right, such increased generality is not required to support the basic notions of exemplars.

With some effort, the former technique can be viewed primarily as a change in terminology; i.e., the class is referred to as a descriptor for an instance and the meta-class as a descriptor for the class. Additional new objects are required to supplant the class objects that became descriptors. However, the approach is unsatisfying because the information logically associated with the exemplar is in a distinct object.

The second technique removes this objection and associates the information directly with the exemplar (and in fact, any instance cloned from that exemplar). It does require the introduction of object subparts visible by the garbage collector but inaccessible as distinct Smalltalk objects. A change in the compiler is also required to make the additional fields visible and accessible and to generate new bytecodes that load and store indirect. This latter technique is being used in Actra.

7 Making Smalltalk Exemplar Based

The inheritance mechanism required to support an exemplar based Smalltalk is equivalent to the class based inheritance mechanism except that inheritance takes place through exemplars rather than classes. As shown in Figure 8, for example, exemplar E2 with additional instance variables B1 and B2 (its **local** representation) inherits instance variables A1, A2, and A3 from exemplar E1.



Simple Exemplar Inheritance

Figure 8

To convert to an exemplar based system, changes to all three components of the existing class based system are required: the image, the virtual machine, and the Smalltalk library.

Image Changes: Remove all class instance variables in order to ensure that all classes have a uniform representation. Change each class to an exemplar descriptor and create an instance exemplar for each descriptor (there is no need to create a new one if one already exists). Additional new objects are required to supplant the class objects that became descriptors. Similarly, change each meta-class to a descriptor for each class. Each descriptor requires additional fields to reference the owner (the exemplar) and the class to be associated with the exemplar. As explained above, the descriptor is not directly accessible as a separate Smalltalk object. Dictionary Smalltalk should be expanded to include both named instances and classes; e.g., both Object and anObject, Set and aSet, ... should be available.

Virtual Machine Changes: Implement the special bytecodes (load and store indirect) that provide access to the descriptor information. Direct kernel references to class objects must be modified to accommodate the shift to exemplars; e.g., the **new** primitives should be modified to perform **clone** operations.

Smalltalk Library Changes: The compiler must be modified to provide direct access to the descriptor information from arbitrary objects. Methods previously inherited by classes can then be recompiled as methods associated with anObject. Similarly, methods previously inherited by metaclasses should be recompiled with Object. Careful changes must done on these methods to separate the strong coupling that exists between classes and meta-classes. **Clone** methods must be added to anObject to enable duplicate objects to be created; the **new** methods in the class must be modified to clone its standard instance exemplar. The browser can be modified in a simple way by having the switch that distinguishes between instances and classes present instance exemplars (anObject, aSet, ...) as disctint from class exemplars (Object, Set, ...). Selective textual name changes will also be required from 'class' to 'exemplar'; e.g., method names **superexemplar** and **subexemplars** instead of **superclass** and **subclasses**, etc.

Perhaps the greatest difficulty is coordinating the above changes into a manageable order. Because it is a bootstrapping process, a staged approach is required.

8 And-Inheritance: (Traditional Multiple-Inheritance)

Multiple-inheritance is a technique for the management of objects constructed by combining two or more different kinds objects (the **super-objects**). For future comparison with a technique to be discussed later, we will call this **and-inheritance**. The composite object has the instance variables of all existing super-objects in addition to extra instance variables introduced locally (the **local representation**). In the existing scheme for multiple inheritance in Smalltalk, conflicts arise when distinct instance variables or methods in different supers have the same name. An instance variable conflict requires a name change to resolve it. A method conflict is resolved by creating a local method that overrides the conflicting methods. Typically, a specific method (or combination of methods) is copied by the user to construct the new local method.

The usefulness of multiple-inheritance is supported by several existing systems. For example, Traits [Curry 84], Loops [Bobrow 84], and Flavors [Weinreb 80] use it extensively. Although Smalltalk supports it [Borning 82], it is a relatively recent introduction; e.g., no existing classes are currently defined using multiple-inheritance. It could have been used, for example, to define ReadWriteStream by inheriting from both ReadStream and WriteStream. Unfortunately, the current implementation is relatively unwieldy since methods inherited from classes other than the primary super must be physically re-compiled in the new environment (actually defeating one of the primary reasons for inheritance -- code sharing).

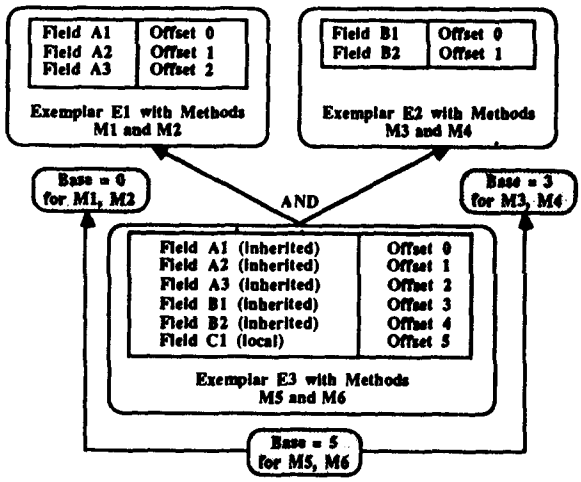
In Smalltalk, the layout of the instance variables is managed using a **contiguous** representation; i.e., fields for the accessible instance variables are stored in neighbouring storage location. By contrast, other systems such as Traits [Curry 84] use a **distributed** or non-contiguous representation. More flexible systems [Lieberman 86] are likely to use both approaches.

A contiguous representation that concatenates the respective instance variables can be made to work without copying or recompiling methods by

1. insisting that local methods have access only to the local representation,
2. associating a unique **base** with each method accessible from the new exemplar, and
3. providing a repository for this base in active contexts.

The first requirement actually eliminates instance variable conflicts because instance variables of inherited exemplars cannot be directly manipulated, even though they exist in the object. Additionally, this property ensures that exemplars are

provided with a greater degree of modularity than existing Smalltalk objects. This deviance from the Smalltalk semantics is expected to have little impact -- it can always be circumvented by defining explicit accessing methods.



Multiple Inheritance (And-Inheritance)

Figure 9

The second and third requirements provide the mechanism for "position independent" references to instance variables. The **base** is the offset for that portion of the representation accessible by the method; i.e., the local representation. In Figure 9, exemplar E3 inherits from both exemplars E1 and E2. The figure illustrates that with respect to exemplar E3 and any of its clones, M1 accesses its instance variables using base 0, M3 using base 3, and M5 using base 5 (to choose a few typical cases). Although not explicitly shown, E2 and its clones would be using a different set of bases; e.g., M3 would be using base 0.

Because the bases are not unique to the methods, a small tree structurally isomorphic to the inheritance structure (a dictionary tree) must be maintained with each exemplar to record the distinct bases. Intuitively, each node of this tree corresponds to one exemplar in the inheritance hierarchy and contains both a base and the method dictionary associated with that exemplar. For the simple inheritance case, the tree for a particular exemplar can share the super exemplar's tree. In general, the amount of storage needed to store this information is negligible if a suitable sharable structure is used.

When method lookup is performed, the base associated with a given method is extracted and stored into the corresponding active method context to enable correct access to the local representation. Instance variables of the exemplar are then accessed by adding the instance variable offset to the base of the method. The method lookup for **super** messages begins with a search in the exemplar containing the method (each method can specify the unique exemplar with which it is associated) and simply adds the new base found to the current base to obtain an updated base.

The **contiguous** representation does not successfully handle cases involving exemplars with varying numbers of fields. Such representations are used for representing arrays and collections, for example. Consider the situation of simple exemplar inheritance as shown in Figure 8 but where both the inherited exemplar E1 and the new exemplar E2 have a varying number of fields; e.g., one collection-like object is trying to inherit from another. This **contiguous** representation is too simplistic to accomodate more than one varying length field in a new exemplar (at the end). Of course, it could be generalized but the result would require substantial modifications to the Smalltalk virtual machine code for field accessing.

A **distributed** representation can be used to circumvent this problem. In Figure 8, rather than expanding the fields of the inherited exemplar E1 into the new exemplar E2, an explicit instance of E1 can be created and a pointer to that instance kept in E2 (instead of the actual instance variables). E2 is then viewed as a composite object with **subpart** E1. As the method lookup mechanism climbs the inheritance hierarchy, the corresponding subpart structure is tracked. When the desired method is found, it is executed on the corresponding subpart.

The approach is made compatible with the above technique that maintains bases by using constant zero for the base. References to self must be resolved by locating the outermost containing exemplar. This can be done by keeping **outermost-self** in addition to the current base in the active context . Messages sent to arbitrary objects must create a new outermost-self (the receiver); messages to self maintain the existing one. If the **push receiver** bytecode is revised to push outermost-self, all message sends can be handled uniformly (the first case). Instance variable access, on the other hand, must use the local self.

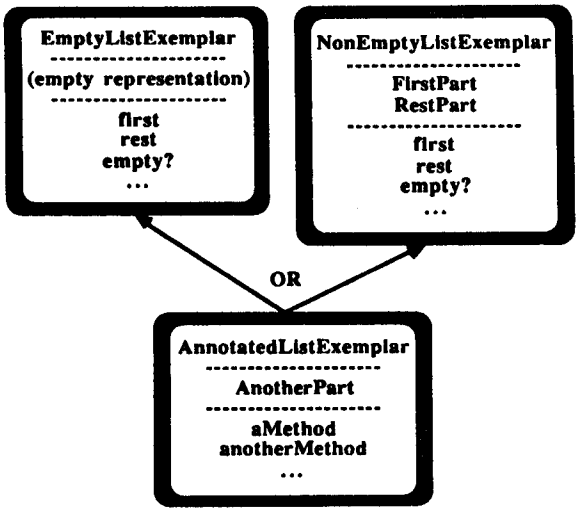
In our initial implementation, we are using only the **contiguous** representation. Since the previously mentioned problem situations are not currently handled by the existing Smalltalk implementations (by design), this approach will maintain the existing functionality while minimizing the impact to the existing implementation

Obvious changes are needed in the method lookup mechanism to handle sharable trees of methods/bases (and the corresponding arithmetic to propagate bases) and in the method send primitives to encode the base and the current exemplar into the active context. Instance variable access must also be relative to the base associated with the active context. Super messages must also be modified to accommodate the above. Note that this implementation strategy can be applied to both class based and exemplar based Smalltalk systems.

8. Or-Inheritance: A New Dimension

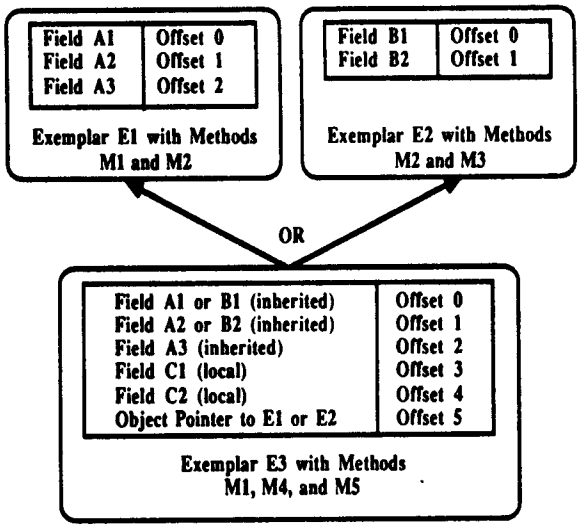
The additional ability to associate several distinct instance exemplars with a class (the multiple representation capability) introduces the potential and perhaps even the need for a new mode of sharing.

To introduce the new mode, recall the example of Lisp-style lists with two representations realized by an empty list exemplar and a non-empty list exemplar. Designing a single new exemplar that is intended to inherit from list instances (without being specific) must inherit either from the empty list exemplar or the non-empty list exemplar but not both. The notion is illustrated by Figure 10. Contrast this approach with that suggested by Figure 6 where two corresponding annotated list exemplars (an empty one and a non-empty one) are provided, each inheriting from the corresponding list exemplar.



Illustrating Or-Inheritance

Figure 10



Inheritance with Multiple Representations (Or-Inheritance)

Figure 11

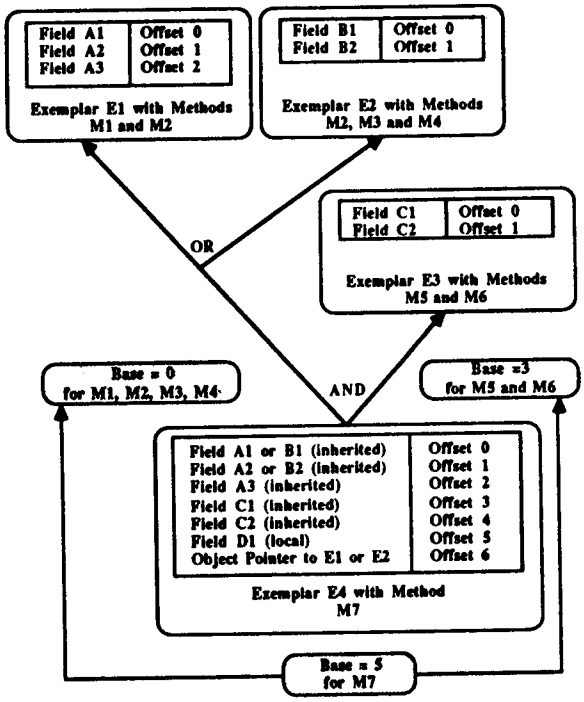
This kind of inheritance (**or-inheritance**) is quite different from the traditional kind of multiple inheritance (**and-inheritance**). Or-inheritance expects the alternative exemplars to have the same methods (distinctions are also permitted) but and-inheritance views such common methods as conflicts that must be resolved.

The fundamental concern in providing an implementation mechanism is the code sharing issue; e.g., multiple instantiations of compiled annotated list methods should be avoided.

The simplest approach is to use a distributed solution. In our example, an annotated list is simply the local representation plus a pointer to one of the two list instances. Cloning an annotated list requires a clone not only of itself but also of the list that is now a subpart. In general, all subparts must be cloned. Without or-inheritance, super-objects in the exemplar are unique; with or-inheritance, selected super-objects are indeterminate (in the descriptor portion) but resolved in the instance itself (the subpart pointer). Method lookup is consequently more complicated. As with and-inheritance, **outermost-self** must be maintained if the object and its subparts is to be viewed logically as one object. This approach is similar to delegation [Lieberman 86] although delegation does not attempt to maintain the "one object viewpoint".

In the absence of and-inheritance, a contiguous representation supporting or-inheritance (see Figure 11) could be made to work by (1) overlaying the instance variables of the inherited alternatives and (2) maintaining an object pointer to the subpart examplar chosen for this specific object. The object pointer in this case is used only for method lookup purposes. Since the instance variables of the subparts are local (the representation being contiguous), there is no need for distinguishing self from outermost-self. The approach is clearly inefficient if the different inherited exemplars have widely differing sizes.

In the presence of and-inheritance, bases are also needed. The combination of both and-inheritance and or-inheritance as illustrated in Figure 12 poses no additional difficulty.



Combined And-Inheritance and Or-Inheritance

Figure 12

9 Conclusions

We have discussed the differences between class based systems and exemplar based systems and shown that the latter have significant advantages. In particular, the exemplar based systems permit the class hierarchy to be divorced from the instance hierarchy. This enables the user's viewpoint to be separated from the implementer's view providing additional fine control of design and implementation considerations. It also permits multiple representations by enabling distinct and arbitrarily different instance exemplars to be associated with the same class.

By converting a class based Smalltalk into an exemplar based system, the original class hierarchy is factored into two distinct hierarchies: the instance hierarchy, since it is the implementation hierarchy, can be maintained as it exists; the new class hierarchy, the logical hierarchy, can be reorganized after the fact to display the more intuitive and expected logical relationship. The new capability permits Set, for example, to be a specialization of Bag and Dictionary to be a generalization of Array while still maintaining the existing representation and methods which are inherited via the instance hierarchy.

We developed the notion of or-inheritance as distinct from and-inheritance and discussed associated implementation issues from the point of view of both contiguous and distributed representations. The Actra project is currently implementing an exemplar based Smalltalk that focuses on contiguous representations.

Exemplars permits evolution in directions that were not previously possible. It also provides clues as to how existing systems like the Lisp Flavour system might be extended to provide sharing both at the source and code level.

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 1984.
2. Borning, A., *The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory*, ACM Toplas, Vol. 3, No. 4, Oct. 1981, pp. 353-387.
3. Borning, A., Ingalls, D.H., *Multiple Inheritance in Smalltalk-80*, Proceedings of the AAAI Conference, Pittsburgh, PA., 1982.
4. Cox, B., *Message/Object Programming: An Evolutionary Change in Programming Technology*, IEEE Software, Vol. 1, No. 1, pp. 50-61, Jan 84.
5. Curry, B., Baer, L., Lipkie, D., and Lee, B., *Traits: An Approach to Multiple-Inheritance Inheritance Subclassing*, Proceedings ACM SIGOA Conference on Office Information Systems, published as ACM SIGOA Newsletter Vol. 3, Nos. 1 and 2, 1982.
6. Duff. C., *Neon - Extending Forth in New Directions*, Proc. of 1984 Asilomar FORML Conf., 1984.
7. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, Mass., 1983.
8. 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.
9. LaLonde, W.R.,*Why Examplars are Better Than Classes*, Technical Report SCS-TR-93, School of Computer Science, Carleton University, May 1986.
10. Lieberman, H., *A Preview of ACT 1*, MIT AI Laboratory Memo No. 625, June 1981.
11. 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.
12. O'Brien, P., *Trellis: Object-Based Environment, Language Tutorial*, Eastern Research Lab., Digital Research Lab Tech. Rep. DEC-TR-373, Nov. 1985.
13. Shapiro, E.Y. and Takeuchi, A., *Object-Oriented Programming in Concurrent Prolog*, New Generation Computing, OHMSHA Ltd and Springer-Verlag, Vol. 1, 1983, pp. 25-48.
14. Tesler, L., *Object-Pascal Report*, Apple Computer, Feb. 1984.
15. 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, pp. 138-140.
16. 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.
17. Weinreb, D., Moon, D., *Flavours - Message-passing in the Lisp Machine*, MIT AI Memo No. 602, Nov. 1980.