AN EXEMPLAR BASED SMALLTALK

David A. Thomas, Wilf R. LaLonde
and John R. Pugh

SCS-TR-94
May 1986

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

This research was supported by NSERC, DREA and DREO.

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 Smalltalk is a rich object-oriented programming environment based on classes and a simple inheritance scheme (though multiple-inheritance is provided non-primitively). The existing organization, however, is overly restrictive. By changing the underlying base to exemplars and still supporting the notion of classes, we show how a much more powerful system can be obtained. In particular, we describe the design and implementation of an exemplar based Smalltalk that not only supports multiple-inheritance (and-inheritance) at the kernel level but also multiple representations (or-inheritance). Additionally, we describe how an existing class-based Smalltalk can be transformed into an exemplar-based Smalltalk.

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. The problems are a consequence of the pivotal role played by classes[LaLonde86]. For instance,

1. Method inheritance and the class hierarchy are intertwined by design making it impossible to separate relationship issues from implementation issues.
2. All instances of a specific class must have identical representations and methods. Thus, (a) instances cannot have specialized methods and (b) multiple representations of instances are not possible.
3. Specializations of classes with individualized representations are not allowed. Subclasses must have a representation that **includes** the superclass representation.
4. Support for multiple inheritance is only provided non-primitively.

When the logical hierarchy and the physical implementation hierarchy are forced to be the same, the physical hierarchy usually wins out. For 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. A future change in the representation, for example, should not entail a change in the logical relationship between classes.

The notion that subclasses must inherit the representation of the superclass is restrictive and logically

incorrect. The class Set, for example, is logically a subclass of Bag since it is a special case in which multiple occurrences of the elements are disallowed. A naive implementation could, of course, 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 current scheme unless the logical hierarchy is changed. In general, the lessons from the data type community emphasize the notion that representation is a dimension totally separable from what can be done with the objects (the operations or methods).

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. Smalltalk itself also supports it [Borning 82] although it is a relatively recent introduction; i.e., no existing classes are currently defined using multiple-inheritance. An example where it might profitably be used is in defining class 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).

As currently implemented, classes are the repository for instance methods and metaclasses are the repository for class methods. The metaclass notion could be discarded altogether if the methods were kept in a more obvious location; i.e., instance methods in an instance and class methods in the class.

In this paper, we describe how a more general and flexible system is obtained by changing the underlying organizational base of Smalltalk from classes to exemplars and by making minimal modifications to the existing organization of Smalltalk. This work is part of a larger implementation effort, the Actra project [Thomas 85]. Actra is a distributed object-oriented computer system based on Smalltalk and targetted for use in industrial applications such as flexible manufacturing, simulation and training, command and control, CAD/CAM and project management.

2. Exemplars As The Base

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 (using the Smalltalk terminology) by

1. a set of instance variables (the **local representation**),
2. an object denoting its class,
3. a set of methods which can manipulate the local representation (the **local methods**), and
4. a set of **super exemplars** (in support of multiple-inheritance).

The terminology is an established one in Artificial Intelligence. At least one programming language exists -- Act/1 [Atkinson 77, Hewitt73-80, Lieberman 81a, b] which uses exemplars as a base. In the long run, the evolution of Smalltalk from a class based system to an exemplar based system is likely to be inevitable.

Exemplars adhere to the principle of modularity to a greater extent than the existing Smalltalk class structure by permitting only local methods to directly access the local representation. This deviance from the Smalltalk semantics is expected to have little impact -- it can always be circumvented by defining explicit accessing methods.

The role of a class is to serve as an interface between users and exemplars of that class. Conventional Smalltalk classes can be viewed as classes with exactly one examplar. Sending a **new** message to a class is equivalent to sending a **clone** message to its standard examplar (in case there is more than one).

3. A Simple Example to Illustrate Exemplars

Without getting into syntactic details, consider the definition of Lisp-style lists with two exemplars: an empty list and a non-empty list. 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 signalling **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 list 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.

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., permitting packed and unpacked representations

without adding to the already large class name space, permitting separate memory-based and disk-based representations -- options that could play an important role as object-oriented databases are developed. In general, the abilitity to provide multiple representation will be of primary importance in industrial applications such as CAD/CAM or flexible manufacturing.

4. Implementation Of Exemplars

Rather than maintain all relevant information about exemplars in a class, an exemplar descriptor fulfills 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 is maintained in the descriptor along with the owner of the descriptor. An exemplar clone is equal in all respects to the exemplar but can be differentiated because it is not the owner of its associated descriptor.

The layout of the instance variables can be managed using either a **contiguous** representation (as provided in the existing Smalltalk system) or a **non-contiguous** representation (as provided by Traits [Curry 84]). Our approach emphasizes the former but permits the latter in exceptional cases. Traits, in contrast, provides only the latter. In the presence of multiple representations and multiple inheritance, the simple approach already used in the existing Smalltalk implementation is inadequate. To simplify the presentation, we consider succesive refinements to the existing system as the above features are added and we deal initially only with the contiguous representation. Then we describe anomalous examples not handled by this representation followed by a discusion of the extensions to support the non-contiguous representation.

4.1 Simple Inheritance

Simple exemplar inheritance, as shown in Figure 1, is equivalent to the simple inheritance mechanism found in standard Smalltalk except that inheritance takes place through exemplars rather than classes. In Figure 1, exemplar E2 with instance variables B1 and B2 (its local representation) inherits instance variables A1, A2, and A3 from exemplar E1.

4.2 Or-Inheritance (Multiple Representations)

Recall the example of Lisp-style lists with two representations realized by an empty list exemplar and a non-empty list exemplar. Designing a 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. 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.

In Figure 2, exemplar E3 inherits (or-inheritance) from a class with two exemplars E1 and E2. The contiguous representation is easily accommodated by overlaying the instance variables of the inherited alternatives. The approach is clearly inefficient if the different inherited exemplars have widely differing sizes.

4.3 And-Inheritance (Traditional Multiple Inheritance)

Multiple inheritance (and-inheritance) occurs when an exemplar simultaneously inherits from instances associated with distinct classes. In Figure 3, exemplar E3 inherits from both exemplars E1 and E2. A contiguous representation that concatenates the respective instance variables can be made to work without copying or recompiling methods by logically associating a unique **base** with each method accessible from the new exemplar. This base is the offset for that portion of the entire representation accessible by the method; i.e., that portion arbitrarily layed out in the total representation for the object. Since only local methods can access the local representation, inheritance from exemplars with the same instance variables is never a conflict.

Figure 3 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 especially 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 combination of both and-inheritance and or-inheritance, as illustrated in Figure 4, poses no difficulty.

4.4 Why The Contiguous Representation Is Not Sufficient?

The **contiguous** representation described above has two disadvantages. In addition to the space inefficiency mentioned earlier, it 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 1 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 **non-contiguous** representation can be used to circumvent this problem. In Figure 1, 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 .

5. Impact To The Existing System

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

The exemplar based system can be derived from the existing class based system by an image transformation, a modification to the virtual machine, and source level changes.

5.1 Image Transformation

The transformation must, as a minimum, perform the following:

- o change each class to an exemplar descriptor and create a standard exemplar for each descriptor (there is no need to create a new one if one already exists).
- o change each metaclass to the class whose methods it contains and add a new field to keep track of its exemplars.
- o change each context object to accommodate two additional fields (the base and current exemplar associated with the compiled method -- needed for handling super).

5.2 Modifications To The Virtual Machine

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 current 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. The **basic new** class primitive must be modified to send a **clone** message to the default exemplar.

5.3 Source Level Changes

A simple change is the addition of a **clone** method to the Object exemplar (to perform a simple shallow copy). More difficult is the revision of the **basic-new** method to send a **clone** message to the default exemplar. Sharable dictionary trees (to contain method/bases) must be defined and integrated with the existing organization (a bootstrapping approach is required). More complex changes include modifying the browser to permit access to all exemplars of a class (although accessing the standard exemplar should be transparent if the above extensions are completed).

As in the existing scheme for multiple inheritance in Smalltalk, method conflict is viewed as an error. Currently, the conflict is removed by requiring the user to define the conflicting method locally. Typically, a specific method (or combination of methods) is copied by the user to create a local method -- not a desirable approach. In future implementations, we aim to provide a simple calculus of methods using set-like operations such as | (or), & (and), and - (not) to indicate inheritance at the method level and an extended super that can refer to specific inherited exemplars. These operations would affect the associated dictionary tree without affecting the corresponding tree for methods higher up the tangled hierarchy.

6. Conclusions

We have described the design and implementation of a multiple-inheritance exemplar based Smalltalk that supports both and-inheritance and or-inheritance. The design permits the existing class system to be supported. However, it also provides substantial enhanced capabilities. For instance, the new design permits the class hierarchy to be divorced from the implementation hierarchy. It will now be possible to make Set a specialization of Bag and Dictionary a generalization of Array without changing the underlying implementation.

Classes in the new scheme are exemplars with their own methods (the class methods) and consequently their own inheritance hierarchy. Instances too are described by exemplars with an inheritance hierarchy. The existing system once transformed will have class and instance hierarchies that correspond one-to-one (a consequence of the fact that classes and metaclasses, except for a few special cases, have mirroring hierarchies). However, this correspondence need not (and should not) be maintained in the new system. The exemplar base 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.

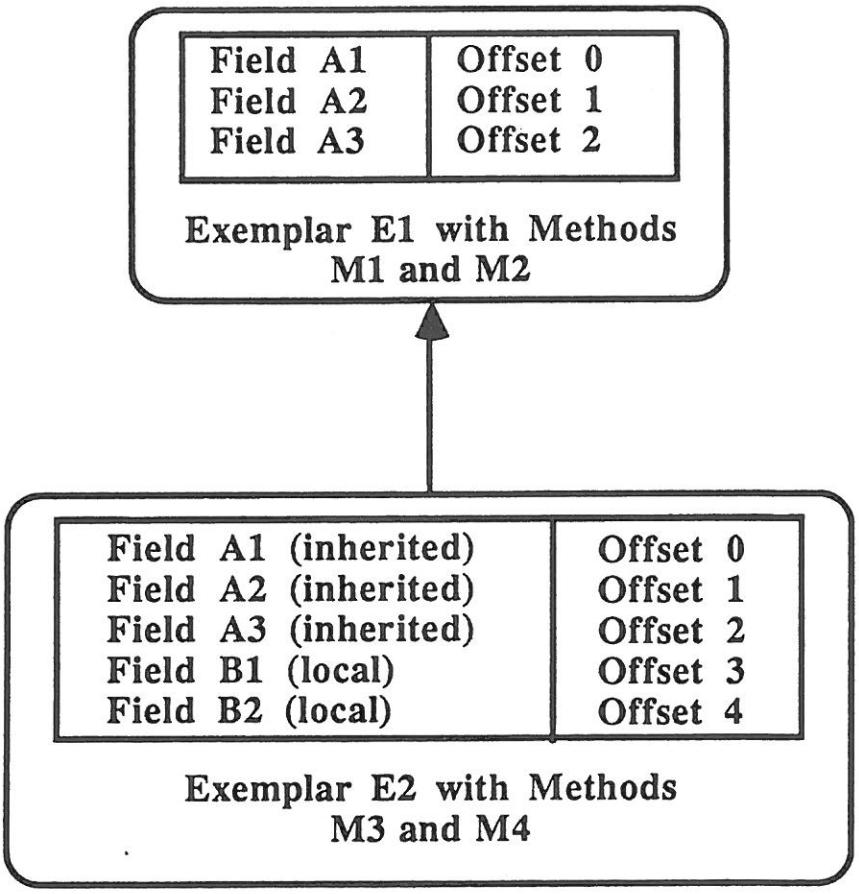
Current Status of The Work

The design for the simpler contiguous representation system is complete and modifications to an existing Smalltalk implementation is in progress. We expect the changes to be complete by May/June 86. Barring unforseen developments, it should be possible to provide performance statistics in a final paper. The final paper should also contain an example that illustrates both and-inheritance and or-inheritance along with a sample run that illustrates the interaction with the different bases. If for any reason the implementation is incomplete, we will be able to emphasize instead the reasons why this new design is better; e.g., by expanding on the conclusions of this paper.

References

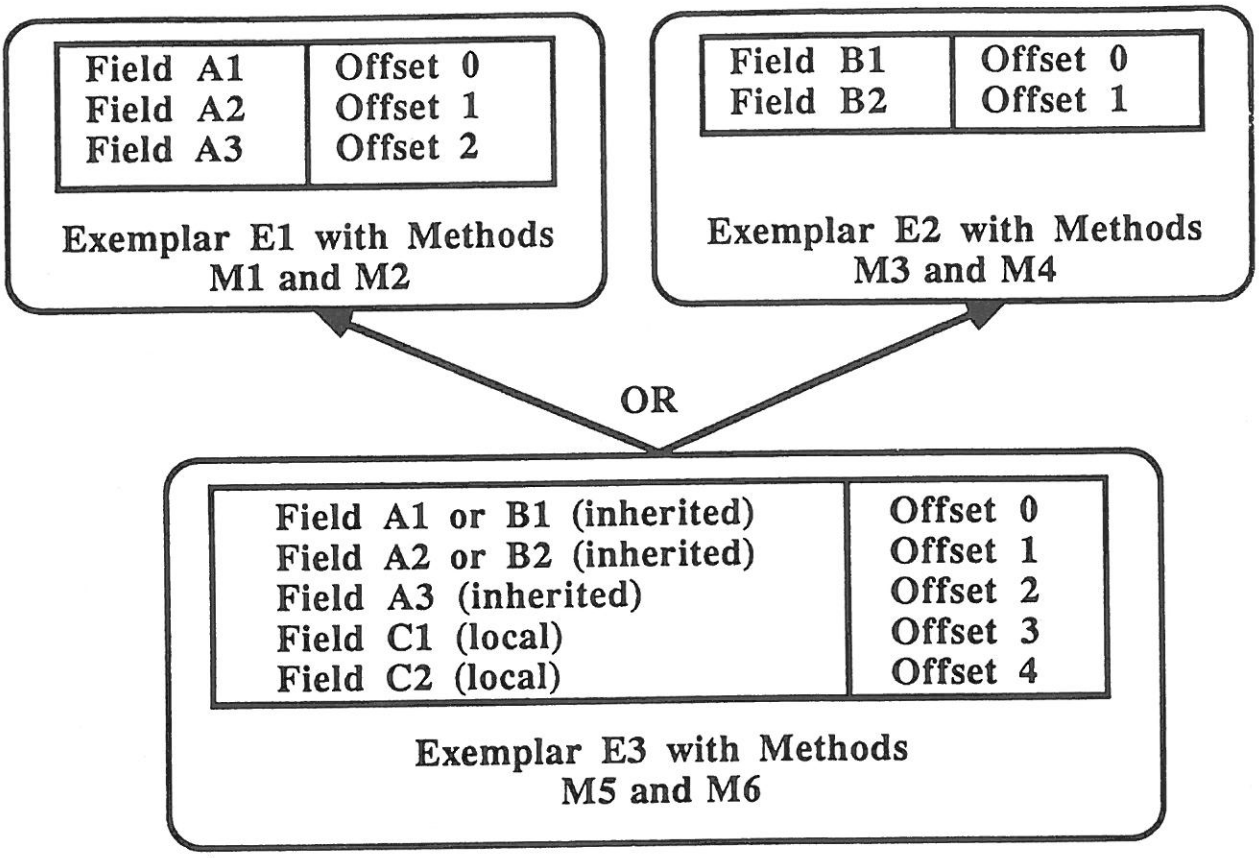
1. Atkinson, R. and Hewitt, C., *Synchronization in Actor Systems*, Conference Record of the 4th. ACM Symposium on Principles of Programming Languages, Los Angeles, California, Jan. 17-19, 1977.
2. Bobrow, D.G., and Stefik, M.J., *The LOOPS Manual (Preliminary Version)*, Knowledge-based VLSI Design Group Technical Report, KB-VLSI-81-13, Stanford University, August 1984.
3. Borning, A., *The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory*, ACM Toplas, Vol. 3, No. 4, Oct. 1981, pp. 353-387.
4. Borning, A., Ingalls, D.H., *Multiple Inheritance in Smalltalk-80*, Proceedings of the AAAI Conference, Pittsburgh, PA., 1982.

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. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, Mass., 1983.
7. Hewitt, C., Bishop, P. and Steiger, R., *A Universal Modular Actor Formalism for Artificial Intelligence.* OJCAI-73 Stanford, California, Aug. 1973.
8. Hewitt, C., *Protection and Synchronization in Actor Systems*, ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication, Santa Monica, California, March 24-25, 1975.
9. Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages*, A.I. Journal, Vol. 8, No. 3, June 1977, pp. 323-364.
10. Hewitt, C., *The Apiary Network Architecture for Knowledgeable Systems*, Conference Record of the 1980 Lisp Conference, Stanford University, Aug. 1980.
11. LaLonde, W.R.,*Why Examplars are Better Than Classes,* Technical Report, School of Computer Science, Carleton University.
12. Lieberman, H., *A Preview of ACT 1.* MIT AI Laboratory Memo No. 625, June 1981.
13. Lieberman, H., *Thinking About Lots of Things at Once Without Getting Confused - Parallelism in Act 1*, MIT AI Laboratory Memo No. 626, May 1981.
14. 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.
15. Weinreb, D., Moon, D., *Flavours - Message-passing in the Lisp Machine.* MIT AI Memo No. 602, Nov. 1980.



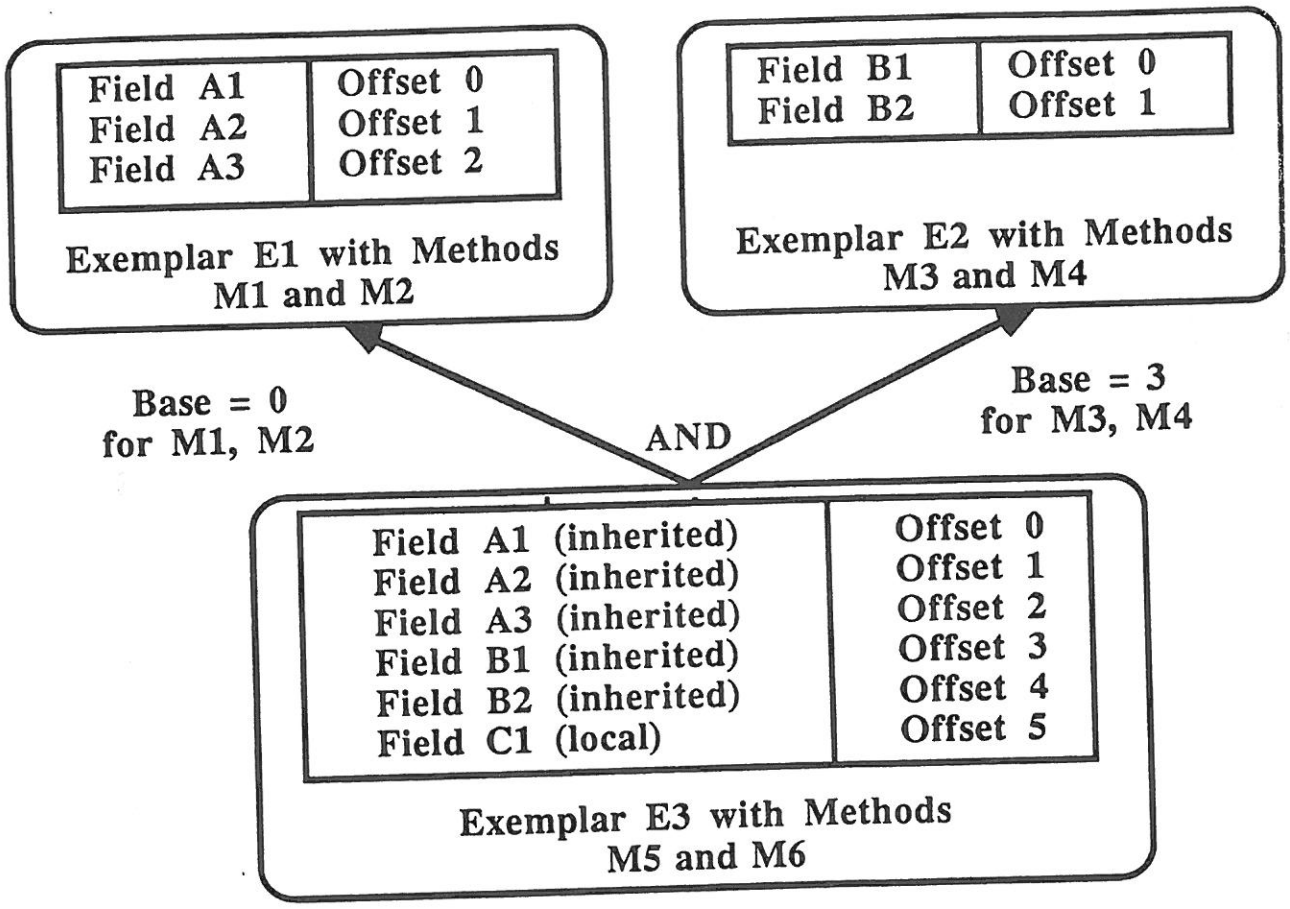
Simple Exemplar Inheritance

Figure 1



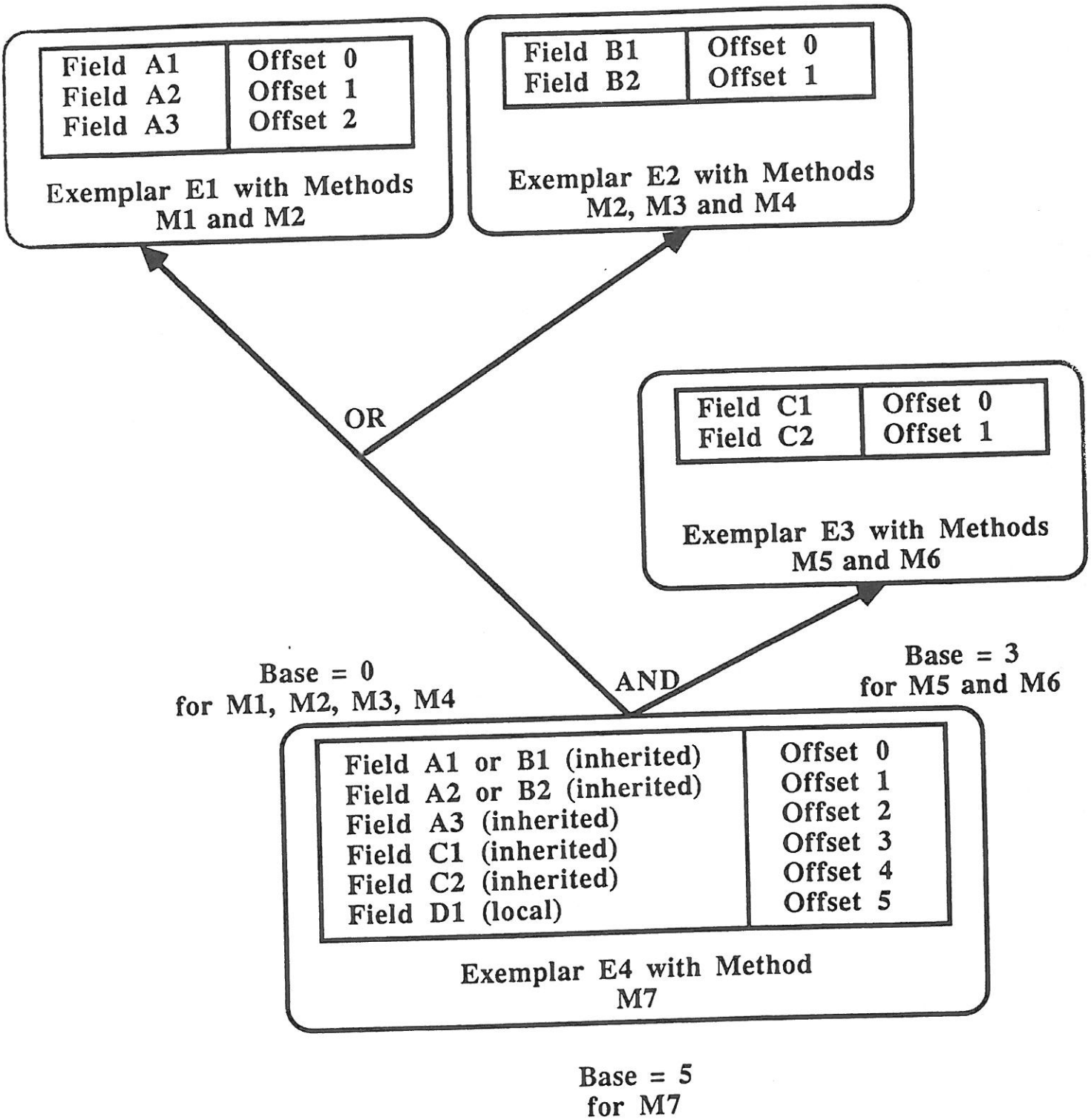
Inheritance with Multiple Representations
(Or-Inheritance)

Figure 2



Multiple Inheritance
(And-Inheritance)

Figure 3



Combined And-Inheritance and Or-Inheritance

Figure 4