Towards a Unified Method of Sharing in Object-Oriented Programming[1]

Lynn Andrea Stein

Technical Report No. CS-89-15

February 1988

[1]This paper appears in *Proc. Workshop on Inheritance Hierarchies in Programming Languages and Knowledge Representation*, Viareggio, Italy, February 1989. The author is supported by an IBM Graduate Fellowship.

Towards a Unified Method of Sharing in Object Oriented Programming

Lynn Andrea Stein*

Department of Computer Science
Brown University

Abstract

The tradeoff between flexibility and behavioral guarantee permeates almost every aspect of computer science. In object-oriented programming, this issue is reflected in the current delegation-vs.-inheritance debate. We describe several mechanisms which impart some of the flexibility of delegation to a more rigidly constrained system. In doing so, we outline a new design methodology which allows the user to program in a more natural and dynamic fashion, occasionally stepping outside the bounds of type inclusion, and then automatically restoring the structure of the strict class hierarchy.

1 Introduction

Design methodologies have traditionally fallen into two opposing camps. Conventional wisdom has it that the best way to design a system is top-down, beginning by sketching the type system on which all else will depend. In object-oriented programming, this corresponds to the class-based approach, in which the class hierarchy functions as the backbone of the system structure. Sadly, this idealized picture of design does not always work in real-world situations, where the type system may not be well-defined at the outset, but rather may emerge during repeated refinements to the project.

Conversely, the design method arising from the need for flexibility and change during early stages is *ad hoc*. All mechanisms for change are available, and it becomes the responsibility of the designer to maintain some semblance of structure throughout, and ultimately to translate his/her understanding of the workings of the system into an orderly type structure. Since no constraints are imposed on definition and construction, no guarantees are possible about the resulting structure. Since no tools or procedures are provided for the translation to a more structured system, the burden of this responsibility is placed squarely on the shoulders of the designer.

*The author is supported by an IBM Graduate Fellowship.

We propose herein the means by which a new design methodology can be constructed. We describe certain controlled extensions to the standard strict hierarchy of class-based object-oriented systems, as well as systematic procedures for restoring these more arbitrary constructs to their traditional counterparts. Thus, a designer has the ability to describe objects in the most natural way, while the system can automatically restore the desired typing guarantees. We believe that these relaxations of the rigid typing required by SmallTalk and other class-based languages will allow much of the flexibility needed in a dynamic and volatile environment, while still providing a framework capable of enforcing the stricter constraints of types.

Historically, these issues have been reflected in the delegation vs. inheritance debate. Proponents of inheritance [Goldberg and Robson, 1983, Schaffert and others, 1986] argued that the class hierarchy was fundamental to the workings of object-oriented systems. In their view, all of the types of objects in the world could be defined and organized in a strictly hierarchical fashion, relatively unchanging over time. Individuals were reduced to mere tokens of these predefined types, differing only in the values of those variables which comprised their states.

In direct contrast, prototype-based languages [Agha, 1987, Lieberman, 1986, Ungar and Smith, 1987] focused entirely on the individual. Objects were defined directly in terms of one another, subject to no overriding constraints. Type became intensional, rather than extensional. Sharing was made explicit. And the need for runtime change was acknowledged through the implementation of delegation through message passing rather than the more traditional copying down and pointer chasing. However, this focus on the individual had its disadvantages as well. Each object became its own definition, and uniformity could only be imposed artificially. Type became, not merely uninteresting, but virtually impossible.

Recent developments in this debate, which has its roots in the philosophical debate over the very meaning of type, have recognized the need for both kinds of relationships.[Stein *et al.*, 1989, Zdonik, 1987] While an explicit, extensional type definition provides much necessary and useful structure, not all objects or situations can be comfortably captured by such a framework. Some relaxing of the type constraints is necessary, to allow anomalous objects and incomplete specifications temporary membership in a generally typed world. But these exceptions should be allowed only at a minimal cost to whatever typing assumptions are otherwise made.

Further, type should be viewed not as purely static, but as potentially dynamic. Given an object at a particular time, its type should be known; however, as object lifetimes lengthen, they may also become increasingly filled with change. Particularly in domains such as CAD and AI, where the world being modeled, and hence the model of the world, are constantly in flux, a static system requiring recompilation with each change is inappropriate and unwieldly.

But while the dynamicism and flexibility described above are desirable properties of a system, neither should they be bought at the expense of systematic and

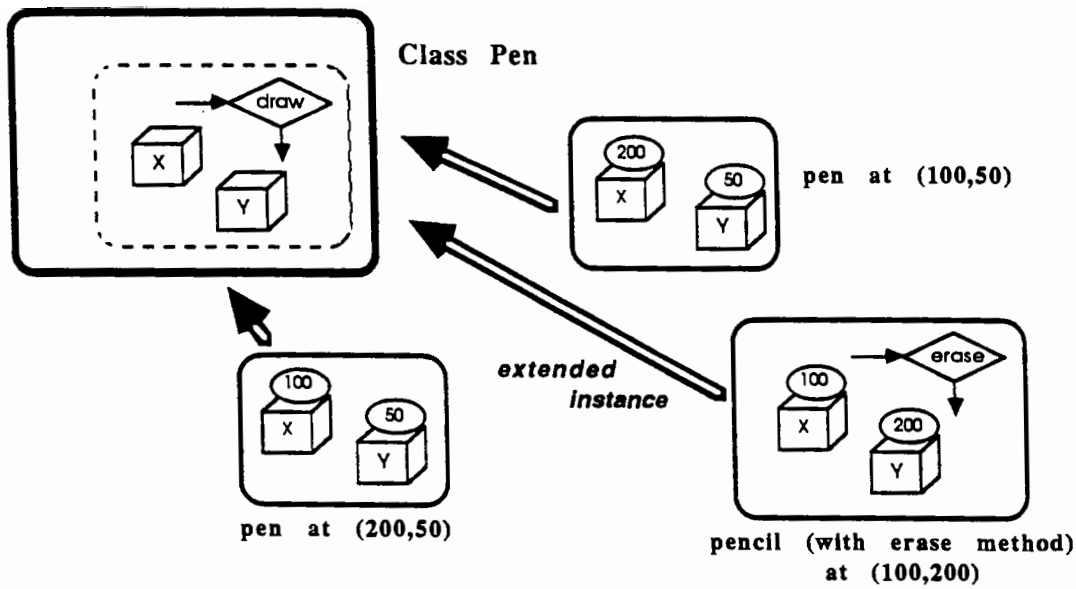

Figure 1: An example of a minimal templates: the `pencil` is an *extended instance* of `pen`.

structural design. Foremost in the principles underlying the extensions outlined below is their ability to be used both in a strictly typed system, to provide the user with perceived flexibility while in fact maintaining a rigid hierarchy, and in a more relaxed system, to allow coherent treatment of anomalies as they occur, as well as reintegration of these novel features into the type system, should this become desirable.

2 Minimal Templates and Extended Instances

Minimal templates were first described in [Zdonik, , Stein, 1987]. They allow an instance to have properties not specified by its class. Thus, a class specifies the fundamental features of its instances, but does not forbid individual variation beyond this specification. Classes therefore provide minimal, but not complete, typing guarantees for these extended instances.

Before discussing the details of the ramifications of such an extension on the type system, let us consider a few examples. The `pencil` example of [Stein, 1987]—shown in figure 1—and the `elephants` of [Lieberman, 1986], are standard AI-like examples, in which these anomalous objects "exist" in the world, and are "discovered" and added to the model. Since real-world individuals vary—this particular `elephant`, and no other, may have a special `bell` or a broken `tusk`, or know an unusual `trick`—it sometimes becomes necessary to add these features to the object representation. However, it is often unwieldly or undesirable to create a special subclass for each possible variation on the combination of attributes potentially or actually possessed by an individual (This is [Lieberman, 1986]'s "one-instance class problem").

But the need to incorporate anomaly is not restricted to artificial intelligence, or to CAD, which has some similar properties. [Moss,] describes a system of persistent objects, some of which possess temporary attributes used only during the running of a particular tool. These attributes (e.g. graphical display information) are not inherently part of the persistent object, and should not, from the user's point of view, be included in its class definition. In addition, the overhead of incorporating them into the persistent object may be substantial; nor are they inherently part of the class definition. Neither, however, should the class of the object be forced to change during the running of the tool. Minimal template provides a systematic means by which the object is allowed to take on these temporary attributes (and, in this context, subsequently to lose them) without violating the constraints of its class. From the user's perspective, this is a desirable state of the world.

The description below of minimal template and its associated mechanisms allow the user this more flexible view of the world. It is, however, unbiased with respect to the question of what the *system* sees. It is possible to implement minimal template, for the user, while actually maintaining strict class-instance relationships. The mechanisms which make this possible are precisely those which allow a system that actually does implement such minimal templates to reintegrate these extended instances and restore strict class membership. Thus, the mechanisms required to understand and use minimal templates are set forth, but the responsibility of appropriate use and integration into an existing system are left to its designers.

From the user's perspective, minimal template may be achieved through the ability to specify additional methods and variables when instantiating a class:

```
pencil = Pen new
            [  x = 50
               y = 200 ]
            [  variables:  ]
            [  methods:
               erase:  ↑ draw white ]
```

This creates an instance of the **pen** class, with an additional *instance* method, **erase**. To the user, the **erase** method appears to be stored in the **pencil** (*instance*) object. Figure 1 provides an illustration.

If instances are allowed to have additional attributes, the typing properties provided by strict class membership are no longer valid. However, unlike a delegation system which is completely untyped, this extension allows *partial* typing. In the above example, while the **pencil**'s **erase** method cannot be typechecked, its **x** and **y** variables, and its **draw** method, can. Such partial typechecking, while not as useful as complete typechecking, allows many advantages without requiring strict adherence to the type hierarchy. In particular, when the classes are well defined, and include the most commonly indexed attributes, the most important typing features, etc., the flexibility of minimal templates can be added with little real impact

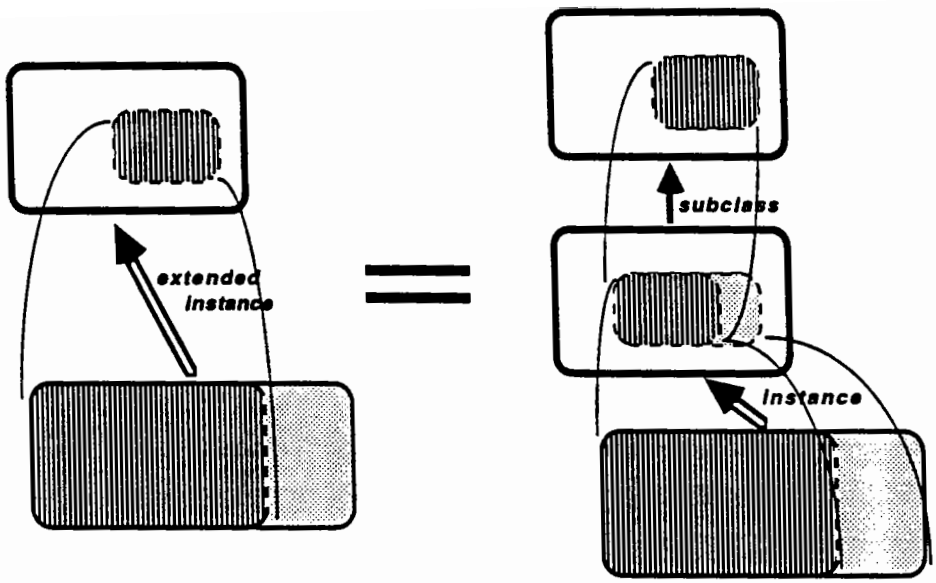

Figure 2: An extended instance is an instance of a (virtual) subclass.

on the typing mechanisms.

Should the user at a later date decide to add a class for this extended instance—perhaps because there are many such objects, and the need for a separate class has become apparent; or because the additional attributes have proved to be persistent, and therefore worth the overhead of adding them to the class; or because these attributes would be useful to typecheck, or to index on—the class for such an extended instance can be automatically generated. In fact, the definitions needed in the instance template of the class are precisely those defined in the `[ method ]` and `[ variable ]` sections of the extended instance definition.

Figure 2 demonstrates this basic truth of minimal templates: *an extended instance is simply an instance of a (virtual) subclass.* That is, anything that can be an extended instance of a class, is really something that could be an instance of a subclass of that class. Minimal templates simply skip the (explicit) definition of the subclass. The partial typing we get with minimal template, then, is precisely the typing we would get by treating a regular instance of a regular class as an instance of a superclass; as long as the two are compatible, it works, but some information is lost. For example, we may chose to treat a `James-Bond-Aston-Martin` as a `Car`, but we lose the ability to index on its many `false-license-plates`, or to use the `ejector-seat`.

To this point, we have discussed the user's perspective. What, now, does the system see? In its most straightforward implementation, minimal templates are viewed in precisely the same way by user and system. The system treats an extended instance in much the same way as any other instance. Should the user wish to *promote* the instance to create a new subclass based on this extension, the system can be asked to do this automatically. Such a strategy is reasonable in a

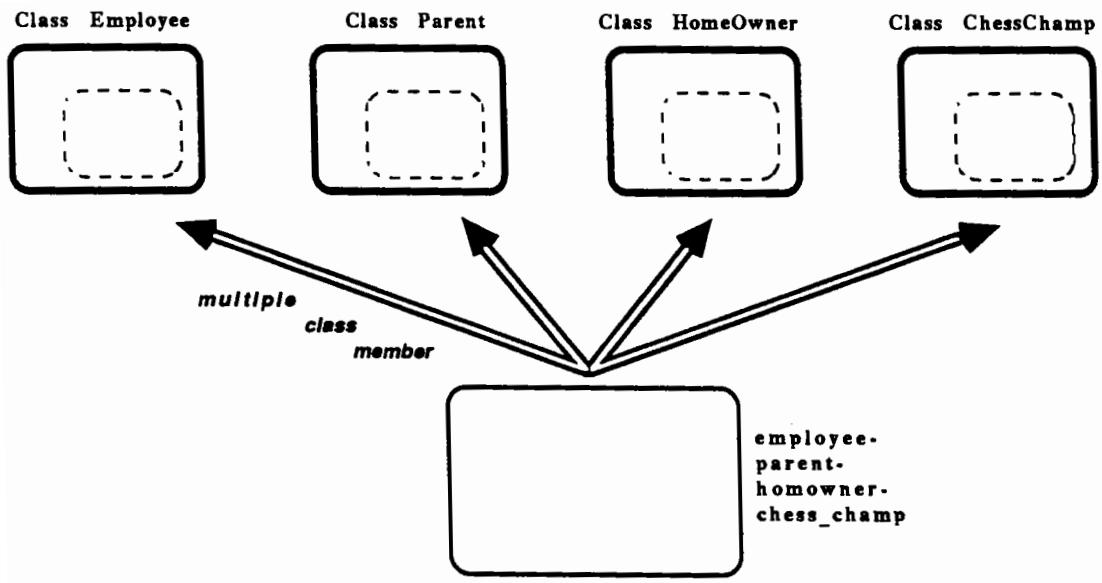

Figure 3: An `emp-par-hom-chc`: an `employee`, `parent`, `homeowner`, and `chess-champion`.

rapidly changing or developing environment, where it is cost-effective to do only that computation which is explicitly required.

However, a system with less dynamicism may have more need for knowledge of and power over types. In this case, minimal templates can be implemented by automatically deducing the type of these extended instances, and actually creating these classes, *internal to the system* and private from the user. The user then sees extended instances, while the system has a standard, strict class membership hierarchy. This implementation is compatible with existing class-based systems, requiring only the additional definition and translation routines to convert minimal template to more standard representations.

3 Multiple Class Membership

In the same way that it is undesirable to create a new class for an anomalous instance, it is undesirable to create a separate class for each cross-combination of any two classes that might occur. For example, a single individual may be an `Employee`, a `Parent`, a `Homeowner`, and a `Chess-Champion`; a `James-Bond-Aston-Martin` is bound to be an `Amphibious-Vehicle` and a `Deadly-Weapon` as well as a `Car`. Yet if there were a single such person, we would not want to create a combined `Emp-Par-Hom-ChC` class, as well as an `Emp-Par-Ren-ChC` class for her brother, who is also a `Chess-Champion` but rents his apartment, and so on for each of the 2^n possible combinations.

The solution, similar to the minimal templates used to resolve the first examples, is to allow a single individual to be a member of more than one class. This corresponds precisely to allowing an instance to do its own multiple in-

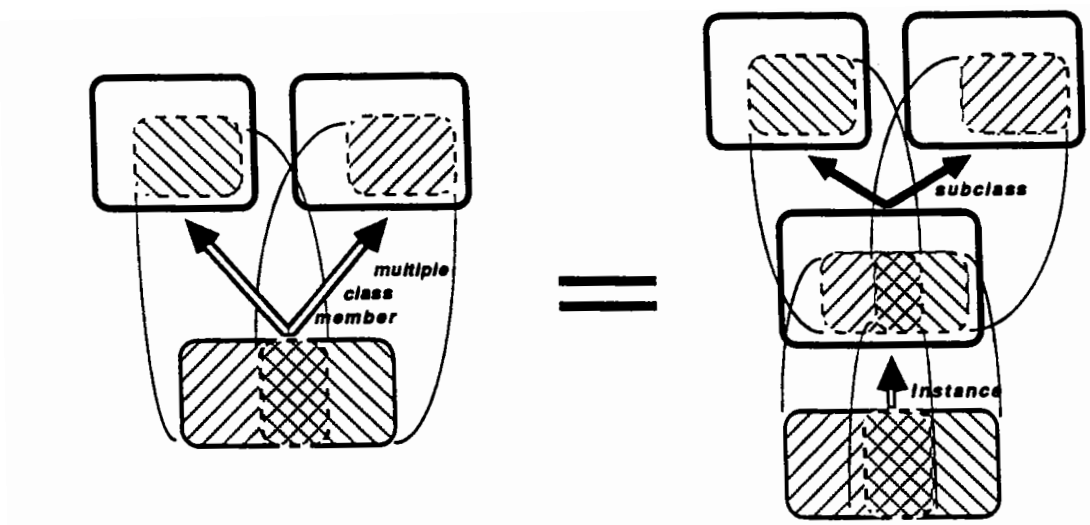

Figure 4: A multiple class member is an instance of a (virtual) multiple-inheritance subclass.

heritance. Figure 3 shows the chessplaying, parental employee described above. Such *multiple class members* are allowed in the PROBE and Iris database systems [Manola, 1987, Fishman and others, 1987].

In some ways, this is more intuitive than actually creating a class of individuals that represent all of those people who work, have children, own homes, and play chess well. In fact, there is probably no occasion on which such a class would be a useful definition to have. On the other hand, employees are regularly processed in standard and relatively well-understood ways. Allowing multiple class membership means that an instance can be treated as an instance of any of its parent classes; it can be typechecked, indexed, and otherwise used in precisely the same way as an extended instance of that class could be. Arguably, these are precisely the properties we want. Extensions to typechecking that would allow it to be typed as an instance of *all* of its classes at once are explored in [Stein and Zdonik, 1989].

The instance which is *a member of multiple classes*, then, *is really an instance of a (virtual) subclass-by-multiple-inheritance* of its classes, as shown in figure 4. It has the same typing properties as an extended instance of any of its parent classes. As with minimal template, this fact can be used in many ways. The system may automatically calculate and interpose the "virtual" class, so that the internal representation includes no violations of strict class membership. Alternately, the system may not compute the intervening classes until such time as the user explicitly requests it.

Since multiple class membership is so closely related to multiple inheritance, it is subject to the same method combination problems described in [Snyder, 1986]. The reduction to subclass relationships provides an effective proof that the solutions to method combination in multiple class membership are precisely those for multiple inheritance, and that those solutions are exactly as efficacious for multiple class

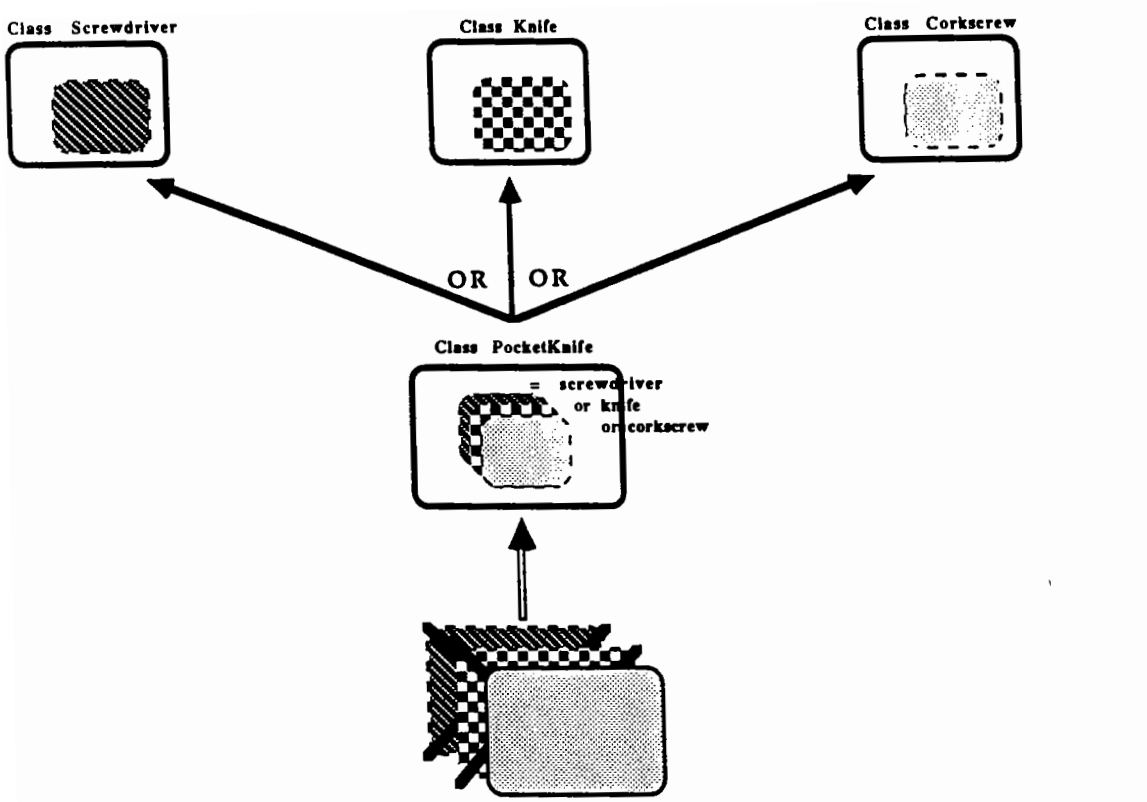

Figure 5: **Swiss-Army-Knives** may be **Corkscrews**, **Knives**, or **Screwdrivers**, depending on which **blade** is in use. This one is currently a **corkscrew**

membership as for multiple inheritance. However, the new perspective provided by multiple class membership may help to provide the intuitions necessary to sort out the conflicts that arise. That is, since a multiple class member is treated as an instance of only one class at a time, rather than all of them at once,[1] the semantics of multiple inheritance can perhaps be understood more clearly in the multiple class membership case.

4 Union Types

The next extension to be considered is that of *union type*. This mechanism, suggested by [Zdonik,] and used in the Trellis/OWL system [Schaffert *et al.*, 1985], is common in block-structured languages such as C and PASCAL. It is a means by which an object can be treated as a member of two (or more) classes, but literally cannot belong to more than once at once. That is, there is an implicit variable in the object, and that variable determines completely the class to which an object belongs.

In some sense, this can be thought of as simply a special kind of method com-

[1]But see below. A multiple class member still possesses at least the union of the methods and variables defined by its classes, while a member of a union class has only the attributes of its current class.

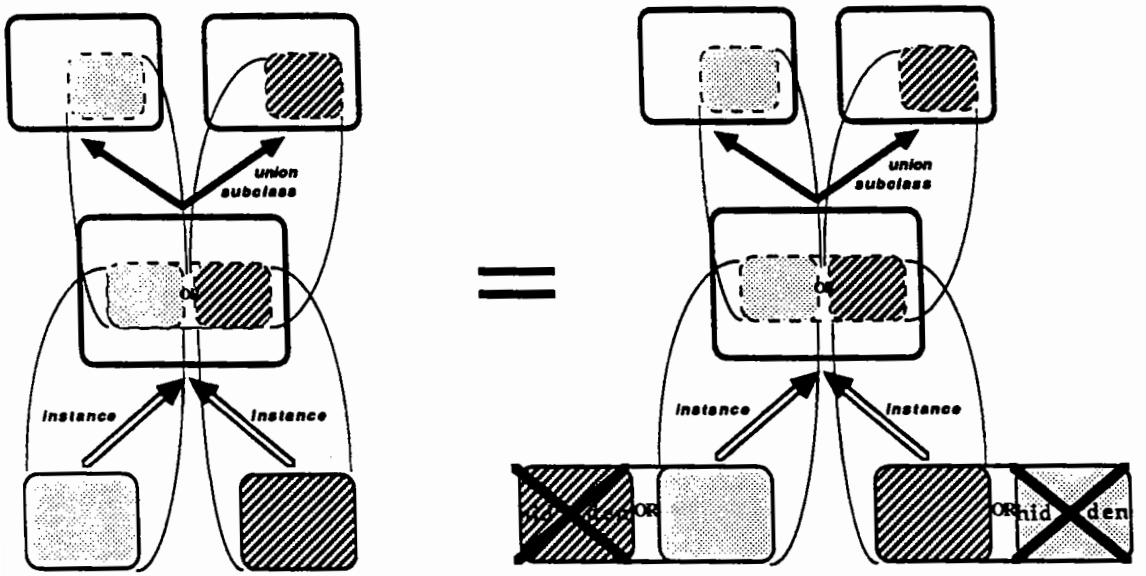

Figure 6: Union types are multiple inheritance subtypes *with only one parent-type visible at a time.*

bination. Depending on the current value of the variable, the object inherits from a particular one of its potential parents. However, unlike a multiple class member, while the object is a member of one class, it does not *at that time* belong to others. Of course, this mechanism can be relaxed (e.g. through minimal template and multiple class membership) so that it in fact provides the same flexibility, but in principle it represents a different concept.

In fact, union type can be implemented at the level of classes. Every instance of a particular class may have the property that it belongs to one of several other classes, and the class to which it belongs may change over time. This is more like traditional multiple inheritance, with the very important exception that the method combination involved, and thus the paths of inheritance, are *dynamic* and subject to change at runtime.

An example is now in order. A `Swiss-Army-Knife` may be a `Knife`, a `Corkscrew`, and a `Screwdriver`. However, it cannot perform the functions of more than one of these at the same time. Its behavior at any given time is determined by which `blade` is in use. Figure 5 shows how minimal templates work for the *class* `Swiss-Army-Knife`, and its instances. By extension, union classes can be combined with multiple class membership to yield union members *without* the intervening union class.

Unlike minimal template and multiple class membership, which reduce cleanly to existing subclass relationships, the statement of union type in terms of static subclassing requires the definition of "concealed" methods and variables. Figure 6 demonstrates the way in which static subclasses can simulate union type. However, systems which allow dynamic (runtime) changes to the hierarchy can capture this

mechanism in a more natural fashion.

5 Conclusion

The mechanisms described above can be incorporated into existing object-oriented systems and paradigms. In adopting these mechanisms, a system provides the user with greater perceived flexibility, allowing more straightforward definitions of relationships and objects. Exceptions are handled in a more natural fashion. The user's sense of how things fit together combines with the system's ability to formalize relationships to create a valuable tool for mapping real worlds onto formal systems. And the most crucial parts of typing are preserved.

There are other uses for these mechanisms as well. In domains where type-checking and indexing have traditionally been difficult or impossible to formalize simply because the types themselves are not known, the more flexible versions of standard object-oriented design tools allow the user to break the type hierarchy in well-defined and controlled ways. To the extent that the core of the type hierarchy is properly designed, the unorthodox additions will have the desirable typing properties formerly so hard to guarantee. As these newer parts of the hierarchy become fixed, complete type definitions can be automatically computed. This leads to a design methodology particularly suitable for the initial stages of CAD or AI, in which the hierarchy is defined in parts, building an intuitive definition, then formalizing it, then extending it intuitively, then formalizing it, in small steps through the cycle of relaxation and tightening of the constraints.

Finally, the mechanisms described above provide a means by which types can be automatically deduced even from a seemingly typeless system such as delegation. However, the transformation from a completely classless system to a wholly class-based system is more or less effective depending on the principles used to build the original prototype hierarchy. Unlike the transformation given in [Stein, 1987], the mechanisms above do not preserve dependence relationships. The properties of a delegation hierarchy which lead to a "good" translation to classes are essentially those given as desirable in [Ungar and Smith, 1987]. Further exploration of the techniques employed there as well as the mechanisms described here will lead to a better understanding of the different uses of delegation and inheritance.

6 Acknowledgements

Thanks are due to Tony Mercado, Peter Wegner, Stan Zdonik, who have been instrumental in helping these ideas reach fruition.

Further exploration of the ramifications of these mechanisms on type theory is described in [Stein and Zdonik, 1989]. In addition, an implementation of a system capable of the tasks described here is discussed in [Mercado Jr., 1988].

References

[Agha, 1987] Gul Agha. *Actors.* MIT Press, Cambridge, Massachusetts, 1987.

[Fishman and others, 1987] D. H. Fishman et al. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Goldberg and Robson, 1983] A. Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Lieberman, 1986] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, Portland, Oregon, September 1986.

[Manola, 1987] Frank A. Manola. Pdm: An object-oriented data model for probe. Draft technical report, Computer Corporation of America, Cambridge, Massachusetts, September 1987.

[Mercado Jr., 1988] Antonio Mercado Jr. Hybrid: Implementing classes with prototypes. Master's Thesis Technical Report No. CS-88-12, Brown University Department of Computer Science, Providence, Rhode Island, 02912, July 1988.

[Moss,] Eliot Moss. Personal Communication.

[Schaffert and others, 1986] Craig Schaffert et al. An introduction to Trellis/Owl. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, September 1986.

[Schaffert *et al.*, 1985] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis object-based environment: Language reference manual. Technical Report DEC–TR–372, Digital Equipment Corporation, November 1985.

[Snyder, 1986] Alan Snyder. Encapsulation and inheritance in object-oriented languages. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon, September 1986.

[Stein and Zdonik, 1989] Lynn Andrea Stein and Stanley B. Zdonik. The dynamic behavior of types and instances. Technical report, Brown University Department of Computer Science, Providence, Rhode Island, 1989.

[Stein *et al.*, 1989] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and Fred Lochovsky, editors, *Object Oriented Concepts, Databases, and Applications*, chapter 3. Addison-Wesley, 1989.

[Stein, 1987] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–146, Orlando, Florida, October 1987.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Orlando, Florida, October 1987.

[Wegner and Zdonik, 1987] Peter Wegner and Stanley B. Zdonik. What is-a is and isn't like. Technical report, Brown University Department of Computer Science, Providence, Rhode Island, March 1987.

[Zdonik,] Stanley B. Zdonik. Personal Communications.

[Zdonik, 1987] Stanley B. Zdonik. Can objects change type? Can type objects change? (extended abstract). In *Proceedings of the Workshop on Database Programming Languages*, pages 193–200, Roscoff, France, September 1987. Altaïr-CRAI.