To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991
© 1991 Kluwer Academic Publishers - Manufactured in The Netherlands

Organizing Programs Without Classes*

DAVID UNGAR†
CRAIG CHAMBERS
BAY-WEI CHANG
URS HÖLZLE (*self@self.stanford.edu*)

Computer Systems Laboratory, Stanford University, Stanford, California 94305

Abstract. All organizational functions carried out by classes can be accomplished in a simple and natural way by object inheritance in classless languages, with no need for special mechanisms. A single model—dividing types into prototypes and traits—supports sharing of behavior and extending or replacing representations. A natural extension, dynamic object inheritance, can model behavioral modes. Object inheritance can also be used to provide structured name spaces for well-known objects. Classless languages can even express "class-based" encapsulation. These stylized uses of object inheritance become instantly recognizable idioms, and extend the repertory of organizing principles to cover a wider range of programs.

1 Introduction

Recently, several researchers have proposed object models based on prototypes and delegation instead of classes and static inheritance [2, 9, 11, 14, 15, 18]. These proposals have concentrated on explaining how prototype-based languages allow more flexible arrangements of objects. Although such flexibility is certainly desirable, many have felt that large prototype-based systems would be very difficult to manage because of the lack of organizational structure normally provided by classes.

Organizing a large object-oriented system requires several capabilities. Foremost among these is the ability to *share* implementation and state among the instances of a data type and among related data types. The ability to define strict interfaces to data types that hide and protect implementation is also useful when organizing

*This work has been generously supported by National Science Foundation Presidential Young Investigator Grant # CCR-8657631, and by Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

large systems. Finally, the ability to use global names to refer to data types and to categorize large name spaces into structured parts for easier browsing are important for managing the huge number of objects that exist in a large object-oriented system.

In this paper we argue that programs in languages without classes are able to accomplish these tasks just as well as programs in class-based languages. In particular, we show that:

- all organizational functions carried out by classes can be accomplished in a very natural and simple way by classless languages,
- these organizational functions can be expressed using objects and inheritance, with no need for special mechanisms or an extralingual layer of data structures,
- the additional flexibility of prototype-based languages is a natural extension of the possibilities provided by class-based systems, and finally,
- exploiting this additional flexibility need not lead to unstructured programs.

The ideas presented here are based on the lessons we learned as we found ways to organize code in SELF, a dynamically-typed prototype-based language [3, 4, 10, 18]. Accordingly, we will illustrate the ideas using examples in SELF, but the ideas could be applied as well to other classless languages providing similar inheritance models.

2 Sharing

Programming in an object-oriented language largely revolves around specifying sharing relationships: what code is shared among the instances of a data type and what code is shared among similar data types.

2.1 Intra-Type Sharing: Classes and Traits Objects

The principal activity in object-based programming is defining new data types. To define a simple data type, the programmer needs to specify the state and behavior that are specific to each instance of the data type and the state and behavior that are common to (*shared by*) all instances of the type. For example, one way to define a simple polygon data type is to specify that each polygon instance contains a list of vertices and that all polygons share an operation to draw themselves.

In a typical class-based language, the class object defines a set of methods and (class) variables that are shared by all instances of the class, and a set of (instance) variables that are specific to each instance. For example, the polygon data type could be implemented by a class `Polygon` that defines a `draw` method and specifies that its instances have a single instance variable named `vertices`; the `Polygon` metaclass would contain a `new` method to create new polygon instances

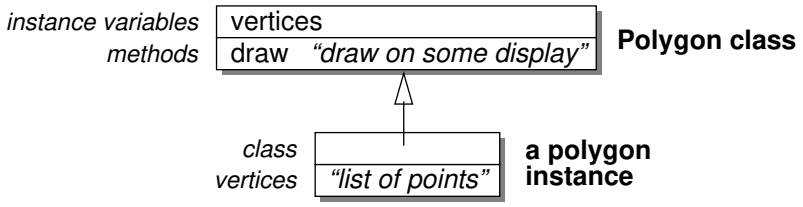

Figure 1a. Data types in a class-based language.

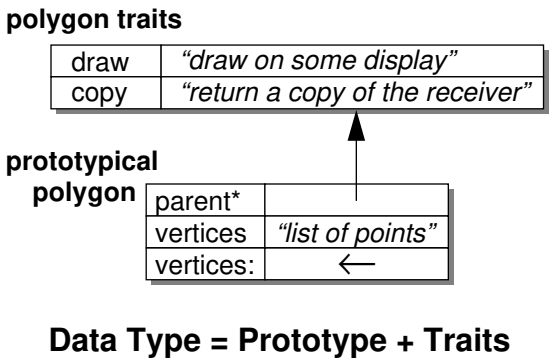

Figure 1b. Data types in a classless language.

(see Figure 1a). To initialize a new instance's list of vertices, the `Polygon` class could define a wrapper method named `vertices:` that just assigned its argument to the `vertices` instance variable. This wrapper method is required in languages like Smalltalk-80[1] [7] that limit access to an object's instance variables to the object itself.

In a classless language, the polygon data type is defined similarly. A prototypical `polygon` object is created as the first instance of the polygon type (see Figure 1b). This object contains three slots: an assignable data slot named `vertices`, the corresponding `vertices:` assignment slot,[2] and a constant *parent* slot[3] pointing to another object that contains a `draw` method and a `copy` method. The

[1]Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

[2]Data slots in SELF may be *assignable* or *constant*. Data slots are assignable by virtue of being associated with an *assignment slot* that changes the data slot's contents when invoked. The assignment slot's name is constructed by appending a colon to the data slot's name.

[3]Parent slots are indicated in SELF syntax by asterisks following the slot name. Parent slots in the figures of this paper are therefore indicated with asterisks.

`vertices` slot in this prototype is initialized to a convenient default list of vertices (e.g., a list of three points defining a triangle), making it usable as is and thus serving as a programming example.

New polygons are created by sending the `copy` message to an existing polygon (such as the prototypical polygon), which first *clones* (shallow-copies) the receiver polygon and then copies the internal vertex list. Since the prototype's slots contain default values, clones of the prototype are automatically initialized with these values as well. In particular, the same parent object is shared by each new polygon, providing the common behavior for all polygons in the system. We call these shared parent objects *traits objects*. Traits objects in a classless language provide the same sharing capability as classes, and just as in a class-based language, making changes to the behavior of all instances of a type is simple since the common behavior is factored out into a single shared object.

In general, data types may be defined in a classless language by dividing the definition of the type into two objects: the prototypical instance of the type and the shared traits object. The prototype defines the instance-specific aspects of the type, such as the representation of the type, while the traits object defines common aspects of all instances of the type. No special language features need to be added to support traits objects—a traits object is a regular object shared by all instances of the type using normal object inheritance. Since traits objects are regular objects, they may contain assignable data slots which are then shared by all instances of the data type, providing the equivalent of class variables.

Classless languages actually gain some descriptive power over class-based languages by dividing the implementation of a data type into two separate objects. If the data type is a *concrete* type (i.e. if instances of the data type will be created), then both the traits object and the initial prototype object are defined. If, however, the type is *abstract*, existing simply to define reusable behavior shared by other types, then no prototypical instance need be defined. Alternately, if there is only ever *one* instance of a particular data type, such as with unique objects like `nil`, `true`, and `false`, then the traits object need not be separated from the object at all. Traditional class-based languages implicitly specify both the shared behavior and the format of the class' instances; without extra language mechanisms they cannot distinguish between concrete, abstract, and singleton data types, with a corresponding loss of descriptive and organizational power.

2.2 Inter-Type Sharing: Subclasses and Refinements

Object-oriented languages with inheritance support *differential programming*, allowing new data types to be defined as differences from existing data types. The implementor of a new data type may specify that the type is equivalent to a combination of existing types, possibly with some additions and/or changes. For example, a filled polygon type might be identical to the polygon type, except that

drawing filled polygons is different from drawing unfilled polygons, and a filled polygon instance needs extra state to hold its fill pattern.

In typical class-based languages, a class may be defined as a subclass of other classes. The methods of the new class are the union of the methods of the superclasses, possibly with some methods added or changed, and the instance variables of the new class are the union of the instance variables of the superclasses, possibly with some instance variables added. For example, filled polygons could be implemented by a `FilledPolygon` class that is a subclass of the `Polygon` class (see Figure 2a). The `FilledPolygon` class overrides the `draw` method, and specifies an additional instance variable named `fillPattern` that all `FilledPolygon` instances will have; the `vertices` instance variable is automatically provided since `FilledPolygon` is a subclass of `Polygon`. To initialize a new instance's fill pattern, the `FilledPolygon` class could define a wrapper method named `fillPattern:` that assigned its argument to the `fillPattern` instance variable.

Filled polygons are defined similarly in a language without classes. A new filled polygon traits object is created as a *refinement* (child) of the existing polygon traits object (see Figure 2b). This traits object defines its own `draw` method. To complete the definition of the new data type, a prototypical `filledPolygon` object is created that inherits from the filled polygon traits object. This object could contain both a `vertices` data slot and a `fillPattern` data slot, plus their corresponding assignment slots. (We will revise this representation to avoid unnecessary repetition of the `vertices` data slot in the next subsection.)

In general, a new data type in a classless language may be defined in terms of existing data types simply by *refining* the traits objects implementing the existing data types with a new traits object that is a child of the existing traits objects. Object inheritance is used to specify the refinement relationships, without needing extra language features.

2.3 Representation Sharing: Instance Variable Extension and Data Parents

When defining a data type as an extension of some pre-existing data types, frequently the instance-specific information of the existing data type should be combined with some extra information particular to the new data type, to construct the instance-specific information of the new data type. For example, a filled polygon instance needs both the polygon information (the list of vertices) plus the new filled-polygon-specific information (the fill pattern). Ideally, the new data type wouldn't need to repeat the instance-specific information it inherits from the existing data types, but instead share the information; this would enhance the malleability of the resulting system, since changing one data type's representation causes all data types that inherit from the changed data type to be updated automatically.

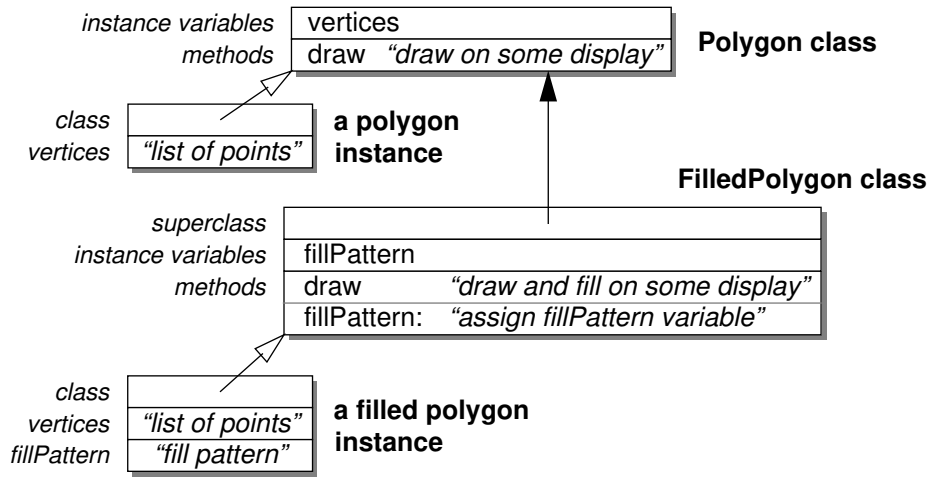

Figure 2a. Differential programming in a class-based language: subclassing, with implicit representation extension.

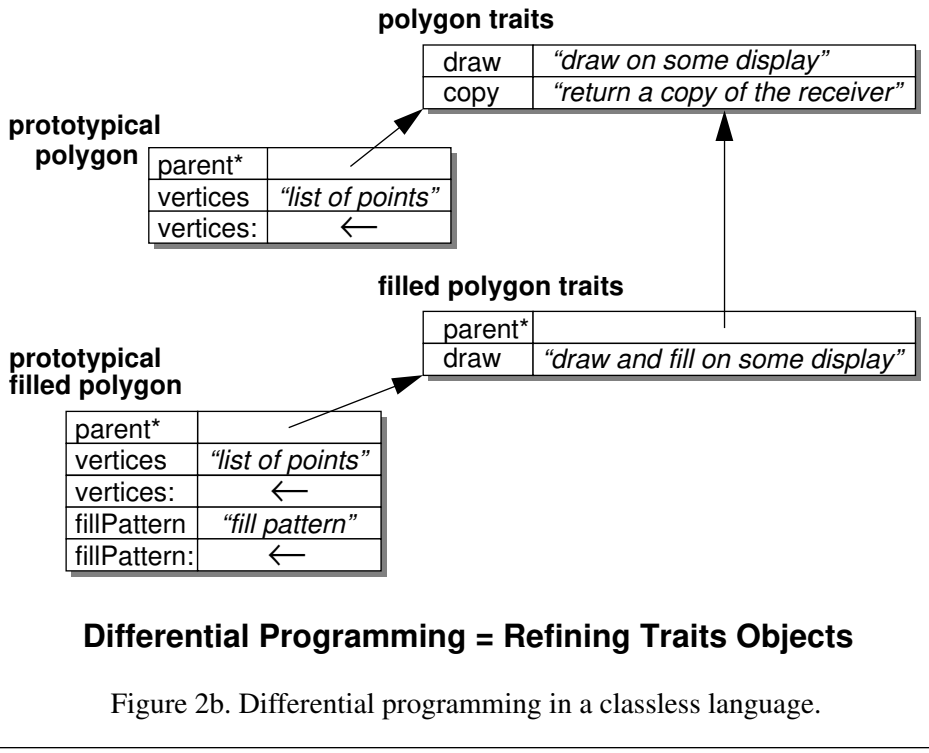

Figure 2b. Differential programming in a classless language.

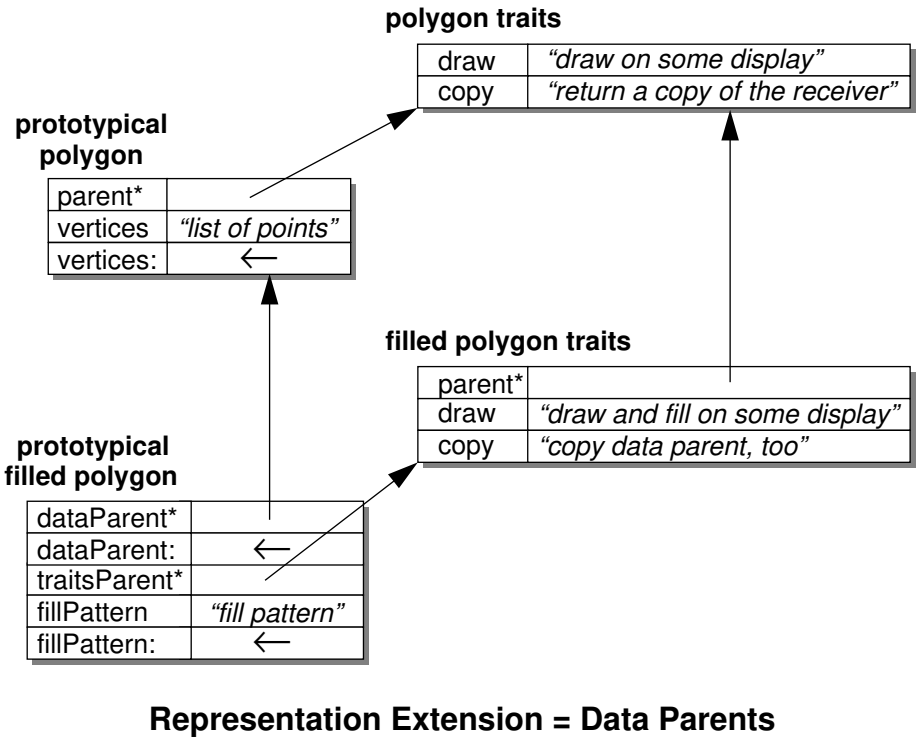

Figure 3. Representation extension in a classless language.

Class-based languages do this well. When a subclass is defined, it automatically inherits the instance variable lists from its superclasses; any instance variables specified in the subclass are interpreted as *extensions* of the superclasses' instance variables. This feature is illustrated by the `FilledPolygon` example class that extends the instance variables of the `Polygon` superclass with a `fillPattern` instance variable (see Figure 2a).

In a classless language with multiple inheritance we can provide similar functionality using *data parents*. Instead of manually repeating the data slot declarations of the prototypes of the parent data types, as was done in the implementation of filled polygons in Figure 2b, the new prototype may *share* the representation of its parent data types by inheriting from them. Thus, a better way to implement filled polygons is to define the `filledPolygon` prototype as a child of both the `filledPolygon` traits object *and* the `polygon` prototype object (see Figure 3). A new `copy` method is defined in the `filledPolygon` traits object to copy both the receiver filled polygon *and* the data parent polygon object, so that each instance

of the filled polygon data type is implemented with two objects, one containing the instance's fill pattern and another containing its list of vertices.

Data parents explicitly implement the representation extension mechanism implicit in traditional class-based languages. Since the data parent objects are parents, data slots defined in the data parent are transparently accessed as if they were defined in the receiver object without defining explicit forwarding methods. By relying only on the ability to inherit state and to initialize a new object's parents to computed values, no special language mechanisms are needed to concatenate representations.

A problem with class-based representation extension surfaces in languages with multiple inheritance. If two superclasses define instance variables with the same name, does the subclass contain *two* different instance variables or are the superclasses' instance variables merged into *one* shared instance variable in the subclass? For some programming situations, it may be correct to keep two different instance variables; for other situations, it may be necessary to share a single instance variable. Different class-based languages that support multiple inheritance answer this difficult question differently; some languages, like C++ [16, 17], provide the programmer the option of doing either, at some cost in extra language complexity.

Classless languages don't face this dilemma. Since the prototypical instance of the data type is defined explicitly, the programmer has complete control over each type's representation. If the new type should contain only one version of the data slot, then the prototype just contains that one data slot. If several versions need to be maintained, one per parent data type, then data parents may be used to keep the versions of data slots with the same name.[4]

2.4 Beyond Representation Sharing

Class-based languages automatically extend the representation of a subclass to include its superclasses' instance variables. However, this automatic extension may not always be desired. For example, an application might want to define a rectangle data type as a subtype of the polygon data type. The representation of the rectangle might be four numbers (instead of a list of four vertices), and the draw routine could be optimized for this special case.

Most class-based languages cannot define such a `Rectangle` class as a subclass of the `Polygon` class because the `Rectangle` class would be extended automatically with the `Polygon` class' `vertices` instance variable. To fix this problem, an additional `AbstractPolygon` class (with no instance variables) must be defined as the common superclass of both `Polygon` and `Rectangle`; the behavior common to all polygons would then be moved from the concrete

[4] SELF includes a message lookup rule (the sender path tiebreaker rule) that automatically disambiguates internal accesses to these data slots.

`Polygon` class to the `AbstractPolygon` class (see Figure 4a). But this then creates another problem: the code for abstract polygons can no longer access the `vertices` instance variable, even for `Polygon` instances. Only instances of `Polygon` and its subclasses know about the `vertices` instance variable. One possible solution would be to define wrapper methods to access the `Polygon` class' `vertices` instance variable from within the `AbstractPolygon` class; the `Rectangle` class would define a method to construct a list of vertices from its four numeric instance variables.

To avoid any problems with altering the representation of a class in a subclass, only *leaf* classes should be concrete and define instance variables. All other non-leaf classes should be abstract, defining no instance variables, and their code should be written to invoke wrapper methods instead of explicit variable accesses. This programming style would support reuse of code while still allowing the representation of a subclass to be different from the representation of a superclass. But it would sacrifice the ability to share representation information by concatenating the instance variables of a class' ancestors, and it would require the definition and use of wrapper methods to access the instance variables. Thus, programs would be more awkward to write and modify.

Prototype-based languages can change the representation of refinements easily. In the rectangle example, the prototypical `rectangle` object contains four data slots and a parent slot pointing to the rectangle traits object, but doesn't include any data parent slots (see Figure 4b). By not including a data parent to the prototypical `polygon` object, the implementation is explicitly deciding not to base the representation of rectangles on the representation of polygons.

The rectangle traits object overrides the polygon traits object's `draw` method with one that is tuned to drawing rectangles using the representation specific to rectangles. To preserve compatibility with polygons, the rectangle traits object defines a `vertices` method to construct a list of vertices from the four numbers that define a rectangle. This is particularly convenient in SELF since a `vertices` message sent in a method in the polygon traits object would either access the `vertices` data slot of a polygon receiver or invoke the `vertices` method of a rectangle receiver, with no extra wrapper methods needed for the `vertices` data slot or modifications to the invoking methods. This convenience is afforded by SELF's uniform use of messages to access both state and behavior, and could be adopted by other classless and class-based languages to achieve similar flexibility. Trellis/Owl [12, 13], a class-based language, also accesses instance variables using messages and is able to change the representation of a subclass by overriding the instance variables inherited from its superclasses with methods defined in the subclass.

An implementation of a data type in a classless language can specify whether to extend the parent types' representations when forming the new type's representation by either including data parents that refer to some of the parent types' repre-

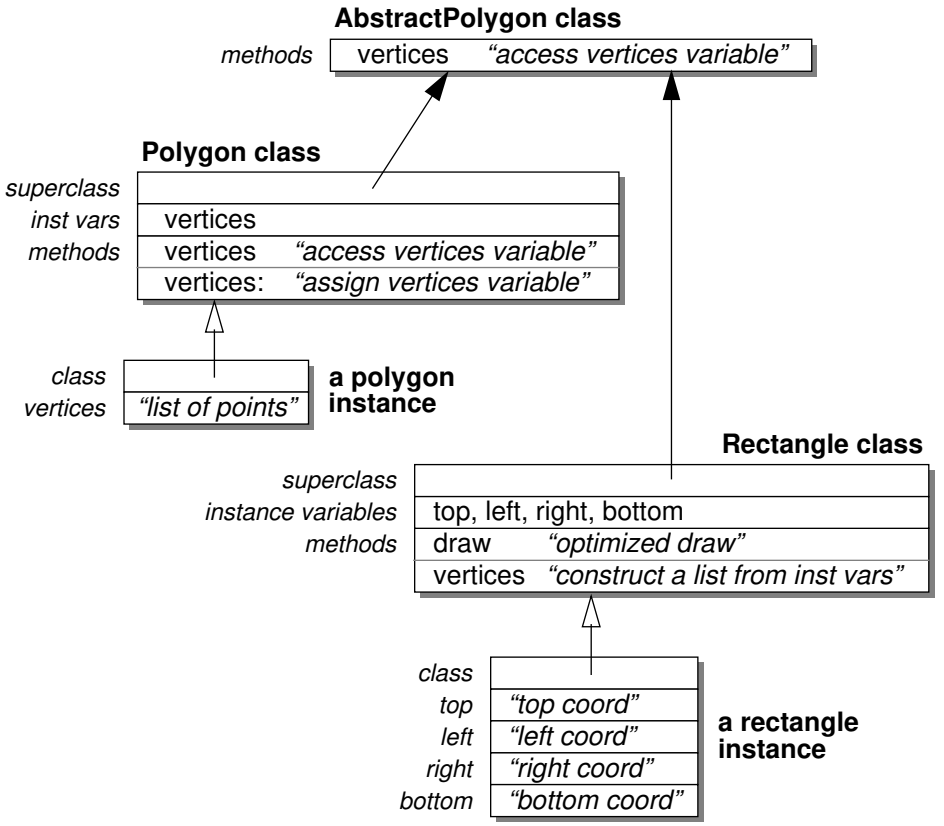

Figure 4a. Representation modification in a class-based language.

sentations (as in the filled polygon example) or not (as in the rectangle example). Both are natural and structured programming styles fostered by classless languages. Class-based languages typically have a much more difficult time handling cases that differ from strict representation extension. As mentioned above, Trellis/Owl is one notable exception. Languages with powerful metaclass facilities, such as CLOS [1], are able to define metaclasses for subclasses that do not inherit the instance variables of their superclasses, but this solution is much more complex and probably more verbose than the simple solution in classless languages.

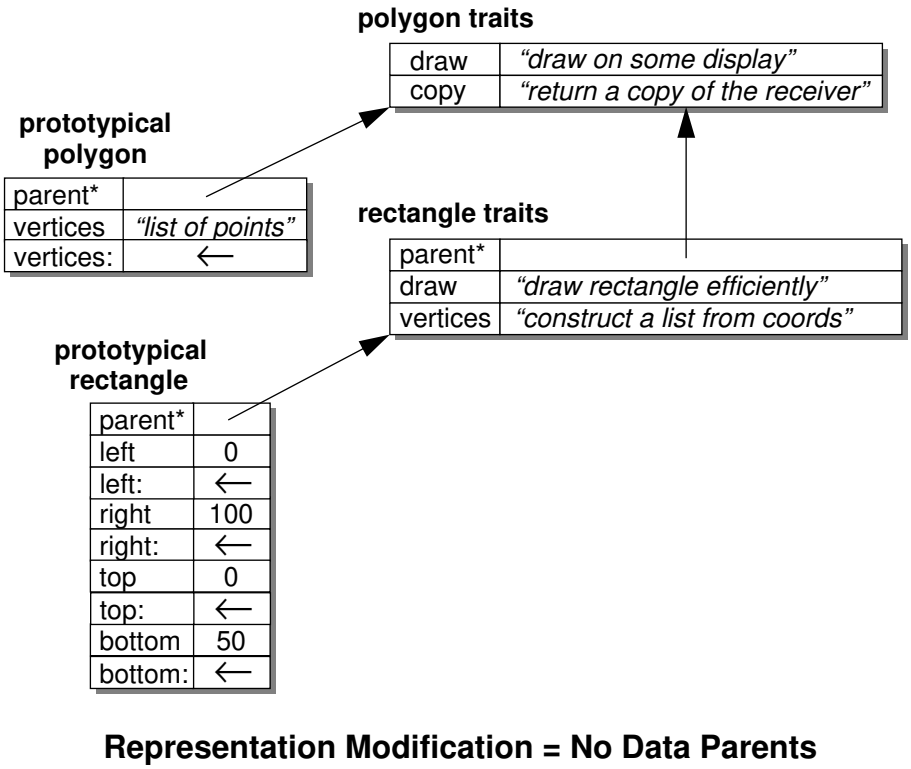

Figure 4b. Representation modification in a classless language.

2.5 Dynamic Behavior Changes: Changing An Instance's Class and Dynamic Inheritance

Sometimes the behavior of an instance of a data type can be divided into several different "modes" of behavior or implementation, with the state of the instance determining the mode of behavior. For example, a boxed polygon (using straight lines) has very different drawing methods than a smoothed polygon (using splines). In many situations, the distinction in "behavior" may be completely internal to the implementation of the data type, reflecting different ways of representing the instance depending on the current and past states of the object. A self-reorganizing collection might use radically different representations depending on recent access patterns, such as whether insertion has been more or less frequent than indexing, even though the external interface to the collection remains unchanged.

One common way of capturing different behavioral modes is to include a flag instance variable defining the behavior mode, and testing the flag at the beginning

of each method that depends on the behavior mode. This obscures the code for each behavior mode, merging all behavior modes into shared methods that are sprinkled with if-tests and case-statements. This code is analogous to programs simulating object-oriented method dispatching: if-tests and case-statements are used to determine the type of the receiver of a "message." Not surprisingly, flag tests for behavior modes suffer from the same problems as flag tests for receiver types: it is hard to add new behavior modes without modifying lots of code, it is error-prone to write, and it is difficult to understand a particular mode since its code is intermixed with code for other behavior modes.

A better way of implementing behavior modes is to define each mode as its own special subtype of the general data type, and use method dispatching and inheritance to eliminate the flag tests. For example, the collection data type could be refined into an empty collection data type and a non-empty collection data type, using inheritance to relate the three types [8]. However, the behavior mode of an instance may change as its state changes: an empty collection becomes non-empty if an element is added to it. This would correspond in a class-based language to changing an object's class dynamically, and in a prototype-based language to changing an object's parent dynamically.

Most class-based languages do not allow an object to change its class, and those that do face hard problems. Since the class of an object implicitly specifies its representation, what happens to an object that changes its class to one that specifies a different representation? An object could be restricted to change its class only to those that have identical representations, but this wouldn't allow different behavior modes to have different representations.

Classless languages, on the other hand, can be naturally extended to handle dynamically-changing behavior modes by allowing an object's parents to change at run-time; an object can inherit from different behavior mode traits objects depending on its state. If the representations of the behavior modes differ, data parents can be used for behavior-mode-specific data slots; changing the behavior mode would then require changing both the traits parent and the data parent (or simply having the behavior mode data parent inherit directly from the behavior mode traits object and changing just the data parent). In SELF this *dynamic inheritance* comes for free with the basic object model. Since any data slot may be a parent slot, and any data slot may have a corresponding assignment slot, any parent slot may be assignable; an object's parents are changed simply by assigning to them.

In the polygon example, the boxed `draw` method would be the same as the `draw` method defined before in the polygon traits object; the smooth `draw` method would treat the vertices of the polygon as the spline's control points. The `polygon` prototype's parent slot would be assignable and alternate between the boxed polygon traits object and the smooth polygon traits object (see Figure 5).

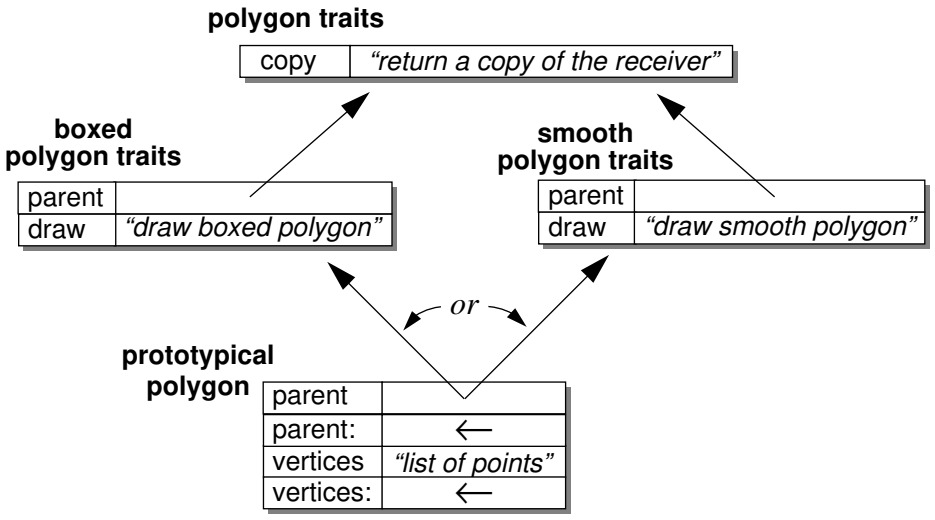

Figure 5. Multiple behavior modes in a classless language.

Behavior modes are naturally implemented in classless languages by using dynamic inheritance to choose from a small set of parents. This style of programming does not compromise the structure of the system; on the contrary, it can make the structure and organization of the system *clearer* by separating out the various modes of behavior. In contrast, the close coupling between a class and its representation prevent class-based languages from being extended naturally to handle behavior modes.

3 Encapsulation

Languages with user-defined data types usually provide a means for a data type to hide some of its attributes from other types. This encapsulation may be used to specify an external interface to an abstraction that should be unaffected by internal implementation changes or improvements, isolating the dependencies between a data type and its clients. Encapsulation may also be used to protect the local state of an implementation of a data type from external alterations that might violate an implementation invariant. Encapsulation thus can improve the structure and organization of the system as a whole by identifying public interfaces that should

remain unaffected by implementation changes and allowing an implementation to preserve its internal invariants.

Existing encapsulation models are based on either objects or types. In languages with object-based encapsulation, such as the Smalltalk, Trellis/Owl, and Eiffel, the only accessible private members are the receiver's. In languages with type-based encapsulation, such as C++, the private members of *any* instance of the type are accessible from methods defined in the type.[5] Type-based encapsulation is significantly more flexible, supporting binary methods that need access to the private data of their arguments and initialization methods that need access to initialize the private state of newly created objects. With only receiver-based encapsulation, these situations require that initialization methods and wrapper methods be in the external public interface to the type, largely defeating the purpose of encapsulation in the first place.

Since classless languages have no explicit classes or types, it would appear that type-based encapsulation would be impossible to support, severely weakening any encapsulation provided by the language. Perhaps surprisingly, SELF's visibility rules *do* support a form of type-based encapsulation [5]. A method may access the private slots of any of its descendants or ancestors, so that a method defined in a traits object may access the private slots of all "instances" of the trait (i.e. clones of prototypes inheriting from the traits object), just as methods defined in a C++ class may access the private members of all instances of the class and its subclasses. In effect, the traits object itself defines a "type," with all descendant objects considered members of the type.

For example, in the polygon example before, the `polygon` prototype object's `vertices:` slot could be declared to be a private slot. This would prevent outside objects from modifying a polygon's list of vertices, but would allow the `copy` method defined in the `polygon` traits object to send the `vertices:` method to the new copied polygon object, since that new object is a descendant of the `polygon` traits object. Similarly, the assignment slots for rectangle objects could also be marked private to prevent unwanted external modification.

Both class-based and prototype-based languages may provide features for encapsulation, even for type-based encapsulation. These features are more dependent on the individual languages than whether the language includes classes or not.

4 Naming and Categorizing

Any system must be structured so that programs can name well-known objects and data types and so that programmers can find objects and types. Objects and

[5] Eiffel includes selective export clauses that allow object-based encapsulation to be extended to type-based encapsulation for particular members.

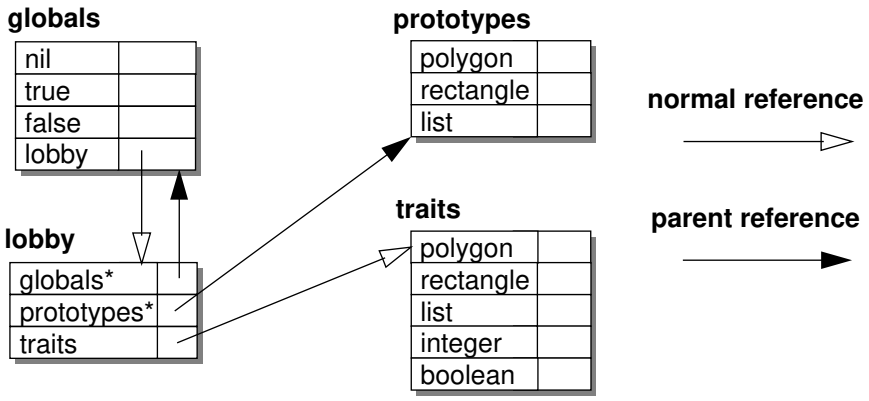

Figure 6. Name spaces are used for global references.

object inheritance support these tasks without explicit support from classes or extralingual environment structures.

4.1 Naming Objects: Global Variables and Name Spaces

Programs need to refer to well-known objects from many different places in the system. For example, a data type may need to be referenced from many places in order to create new instances of the type or to define subtypes. Most class-based languages associate a unique name with each class which may be uttered anywhere in the program to refer to the class; normal instance objects have no explicit names. In a classless language, prototypes and traits objects need to be globally accessible (to clone new objects and to define new refining traits objects), but since these objects are implemented by regular objects, they have no internal names.

In classless languages normal object inheritance may be used to define *name space objects* whose sole function is to provide names for well-known objects. The name of an object in a name space is simply the name of the slot that refers to the object. Any object that inherits the name space object may refer to well-known objects defined by the name space by *sending a message to itself* that accesses the appropriate slot of the name space object.[6] The scope of a name space is the set of objects that inherit from it. The designers of Eiffel encourage a similar strategy to handle shared, possibly global constants, although different language mechanisms are used to handle other global names like class names.

[6]In SELF, this approach is just as concise as global variables because state (e.g. well-known objects) may be accessed using messages without defining wrapper functions, and because messages sent to *self* are written with the *self* keyword omitted. Thus `polygon` is really a message sent to *self* that accesses data; this is just as concise as a global variable access.

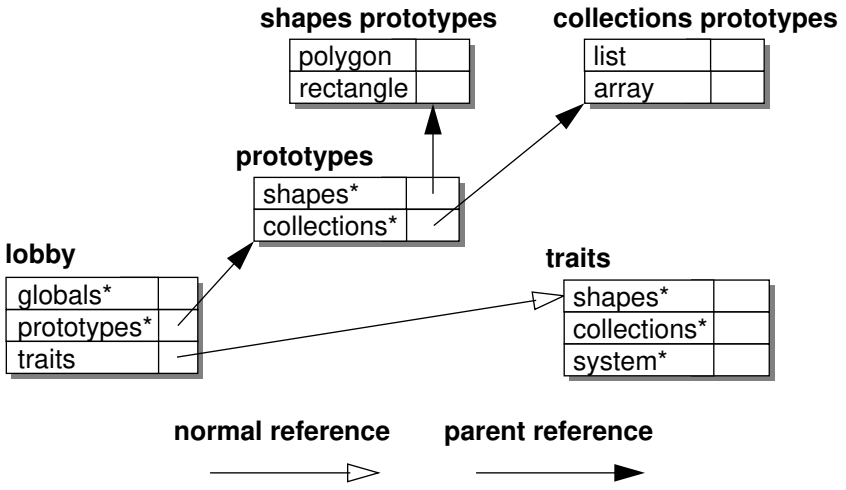

Figure 7. Categorizing name spaces.

Figure 6 illustrates name spaces with part of the inheritance graph in the SELF system. The `lobby` object is the "root" of the inheritance graph, since most objects inherit from it and expressions typed in at the SELF prompt are evaluated with the `lobby` as *self*. The `prototypes` parent object is therefore inherited by most objects and so provides succinct names for the prototypes of the standard data types. The `traits` object contains slots naming the traits objects in the system, typically using the same name as the name for the data type's prototypical instance. For example, the expression `polygon` names the polygon prototype, and the expression `traits polygon` names the polygon traits object. In the first case, since the prototypes name space object is inherited via the lobby, `polygon` yields the contents of the polygon slot in that name space object; in the second case, sending `traits` to the lobby gives the traits name space object, and sending `polygon` to that object gives the polygon traits object.

4.2 Organizing Names: Categories

Large flat name spaces for globals are convenient for programs, but awkward for programmers. Many systems provide features to help organize these name spaces into smaller *categories* of names that break down the name spaces into digestible chunks. For example, the Smalltalk-80 environment [6] supports a two-level structure for browsing classes, dividing up classes into *class categories*.

Classless systems using name space objects can be similarly broken down into categories by subdividing name spaces into multiple parents. For example, the

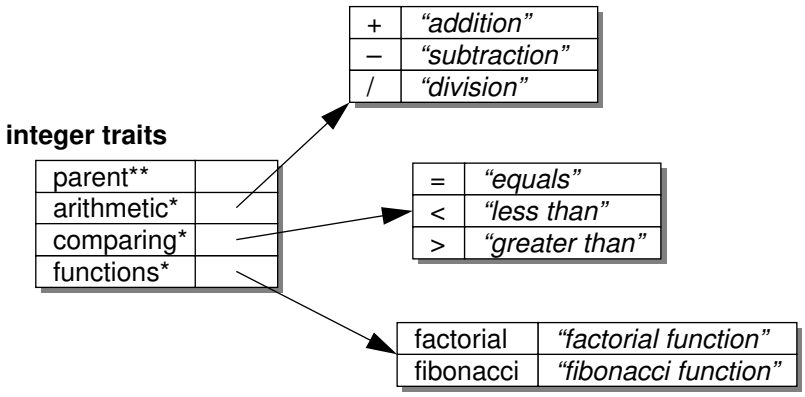

Figure 8. Categorizing traits objects.

`prototypes` name space object could be broken down into several name space subobjects, one for each kind of prototype. The original `prototypes` name space object contains parent slots referring to the name space subobjects; the name of each slot is the name of the category.

These *composite name spaces* behave just like a flat name space from the point of view of the program referring to global objects, since the categories are parents of the original name space object. (For example, the message `polygon` will still yield the prototypical polygon, even when the name space has been broken up into the categories as in Figure 7.) However, since the name spaces are actually structured into multiple objects, the programmer may browse them (using the facilities available for browsing objects) and use both the slot names and the object structure to locate objects of interest and to understand the organization of the system. Composite name spaces may have any number of levels of structure, and need not be balanced (some categories may be subcategorized while others are not). A single object may be categorized in several different ways simultaneously simply by defining slots in multiple categories that all refer to the object. This flexibility is a natural consequence of using normal objects for categorization.

Global variables are not the only name spaces that need to be broken up for programmers. Individual data types are a sort of name space for methods, and these name spaces may be large enough to require their own categorization. The Smalltalk-80 environment again provides a two-level structure for organizing the methods within a class into *method categories*.

For classless languages, the same techniques for organizing large name space objects may be applied to organize large traits objects. Each traits object may refer to parent subobjects that define some category of the slots of the traits object; the

name of each parent slot is the name of that subobject category (see Figure 8). Again these *composite traits objects* extend to any number of levels of structure.

4.3 Extensional vs. Intensional Names and Categorization

By using name space objects and message passing to access global objects, an object's "name" becomes the sequence of message sends needed to reach it. We call this an *extensional name*, since it is derived from the structure of the system. Languages with internal class names, on the other hand, have *intensional names*, since classes are given explicit names by the programmer that may not be related to the structure of the system. Similarly, categorizing name spaces and traits objects using the object structure is *extensional categorization*, while using browser data structures to describe the categorization of classes and methods is *intensional categorization*.

Extensional names have a number of advantages over intensional names:

- No extra language or environment features are needed to support extensional names or categories.
- Extensional names have additional interpretations as *expressions that evaluate to the named object* (and so may be used within a program to access the object) and as *paths to reach the named object* (and so may be used in the browser to navigate to the object).
- The data structures defining intensional names for programmers can become inconsistent with the global variable names used by programs. For example, the internal names for classes and the data structures used by the environment to find a class' subclasses can become incorrect if the global variable referring to the class is renamed or if the inheritance hierarchy is changed without updating the browser's data structures. No such inconsistency can exist with extensional names, since they are derived from the actual structure of the system.

The only restriction associated with extensional names is that they must be legal expressions in the language (since an object's name must be described in the object structure). This restriction has not been a problem in our system, and we feel that the advantages of extensional naming over intensional naming are much more important.

5 Conclusion

Classes are not necessary for structure since objects themselves can provide it: traits objects provide behavior-sharing facilities for their instances and refinements, encapsulation mechanisms can provide type-based encapsulation without needing explicit types or classes, and structured name space objects provide names for programs to use and for people to browse. Traits objects and name space objects

are no different than other objects, but their stylized use becomes an idiom that is instantly recognizable by the programmer. Languages without classes can structure programs as well as languages with classes.

Additionally, certain properties of traditional class-based systems conspire to hinder some kinds of useful structures that are handled naturally by classless systems. Since a class implicitly extends its superclasses' representations, it is hard to define a subclass that *alters* the representation defined by its superclasses. Classless languages define a type's representation explicitly using prototype objects, and so are able to implement both representation extension and representation alteration naturally. Because the representation of an object in a class-based system is so tied up with the object's class, it is difficult to implement dynamic behavior modes. Classless languages may use dynamic inheritance in a structured way to implement these behavior modes as a natural extension of static inheritance. Languages without classes can structure many programs better than languages with classes.

References

1. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification. Published as *SIGPLAN Notices*, 23, 9 (1988).
2. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.
3. Chambers, C., and Ungar, D. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.
4. Chambers, C., Ungar, D., and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 49-70. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
5. Chambers, C., Ungar, D., Chang, B., and Hölzle, U. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. To be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
6. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA (1984).

7. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).

8. LaLonde, W. R. Designing Families of Data Types Using Exemplars. In *ACM Transactions on Programming Languages and Systems*, 11, 2 (1989) 212-248.

9. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.

10. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

11. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.

12. Schaffert, C., Cooper, T., and Wilpolt, C. *Trellis Object-Based Environment: Language Reference Manual, Version 1.1*. DEC-TR-372, Digital Equipment Corp., Hudson, MA (1985).

13. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.

14. Stein, L. A. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 138-146.

15. Stein, L. A., Lieberman, H., and Ungar, D. A Shared View of Sharing: The Treaty of Orlando. In Kim, W., and Lochovosky, F., editors, *Object-Oriented Concepts, Applications, and Databases*, Addison-Wesley, Reading, MA (1988).

16. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA (1986).

17. Stroustrup, B. The Evolution of C++: 1985 to 1987. In *USENIX C++ Workshop Proceedings* (1987) 1-21.

18. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).