OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

An Alternative to Subclassing

David Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon

86-60-2

An Alternative To Subclassing

David Sandberg
Oregon State University

Smalltalk-80 obtains some of its expressive power from arranging classes in a hierarchy. Inheritance is an important aspect of this hierarchy. An alternative organization of classes is proposed that emphasizes description instead of inheritance. This alternative can be used with compile-time type checking and retains the important characteristics of Smalltalk's hierarchy.

1 Introduction

The class-instance model of programming has been used in several languages: Smalltalk-80[3] being the most notable. Some of the power of Smalltalk is obtained by arranging the classes in a hierarchy and using inheritance. Similar hierarchies of subclassing are found in other languages such as the lisp Flavors package[7] and LOOPS[1]. In this paper the function of the class hierarchy in Smalltalk is examined and an alternative hierarchy of classes that focuses on description instead of inheritance is presented.

2 Uses of Subclassing in Smalltalk-80

Smalltalk use a class-instance model of the world. Everything in Smalltalk is an object. Each object is an instance of a class that describes the behavior of the object. The classes themselves are arranged in a class hierarchy. Figure 1 gives part of the hierarchy of classes in Smalltalk. The class *Float* is said to be a subclass of *Number* and *Number* is a superclass of *Float*. The operations of *Number* are automatically inherited by its subclasses.

One use of the class hierarchy of Smalltalk is to factor and share code. For example, the classes for *Float* and *Integer* are subclasses of the class *Number*. *Number* has a message *squared* that is inherited by both instances of the class *Float* and the class *Integer*. The code that implements *squared* is shared between these classes instead of having separate copies.

Smalltalk also uses classes to organize the methods. All the messages for a given class are grouped together with that class. When writing a particular application, it is best to keep all the code for that application together. This is sometimes impossible in Smalltalk. For example, suppose one wanted a new message to integers that returned the n'th Fibonacci number. This message would be added to the other messages for integer instead of being included with the rest of the code for the application. A subclass of integer could be created and the message sent to subclass, but this is awkward.

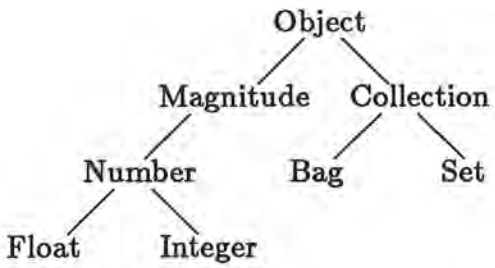

Figure 1: A Class Hierarchy.

Another function of superclasses is to provide abstract descriptions. Additional operations are expressed in terms of the abstract description. Such superclasses are called abstraction superclasses. There are no instances of abstract superclasses. A subclass must be declared that implements the abstract operations. Instances of that subclass can then be used. For example, *Collection* is the abstraction superclass for a group of objects. Two subclasses of *Collection* are *Set* and *Bag*. Each subclass of *Collection* should implement operations to add an element to a collection, to remove an element from a collection, and to iterate through the elements of a collection. Other operations, such as adding two elements to a collection, are implemented in terms of these abstract operations. These other operations are shared among all subclasses of *Collection*. Since there is no syntactic difference between an abstract superclass and any other class, a careful examination of the code must be made to determine if a superclass is an abstract superclass and to find the abstraction operation that must be implemented by the subclass.

Sometimes it is desired to have a subclass have more than one abstract superclass. One example of such a class is the class *Transcript* that displays and records notification messages. It should be a subclass of both *Window* and *WriteStream*. In Smalltalk-80 this is impossible. Schemes have been proposed to allow multiple inheritance in Smalltalk[2]. Such schemes are not totally satisfactory, because the way to merge the inherited behavior from multiple superclasses is not clear and differs from case to case.

Another use of subclasses is to rapidly build classes with a desired behavior. An existing class with a behavior that is close to the desired behavior is found. A new subclass is created and much of the desired behavior is automatically inherited. New methods and instance variables are then added to change the inherited behavior into the desired behavior.

The whole class hierarchy of Smalltalk is based around run-time type checking and run-time binding of messages to methods. If these function are done at compile time, programs are easier to understand, errors are caught earlier, the compiler can perform more optimizations, and the resulting code runs faster.

3 The Alternative

Our alternative to subclassing will use compile time typing, add parameters to classes, and introduce a new form of classes called *descriptive* classes. All these concepts have been fully implemented in an experimental language X2[5,6] and have been used extensively in the user interface to X2. X2 is not used to present these concepts; instead a Pascal like syntax with an informal semantics is used. We assume the reader has a knowledge of parameterized types like those used in CLU[4].

Our alternative focuses on building bigger parts from smaller parts. Most programming languages allow specific parts to be combined. For example, given a specific type, Pascal allows a linked list of that type to be built, but Pascal can not describe how to do this with an unspecified type. For an unspecified type, only the relevant behavior of the parts must by specified while leaving the irrelevant behavior unspecified.

Allowing parameters on classes makes it possible to specify how to construct a linked list whose elements are all instances of the same unspecified class. First a class, *list(f)*, is declared with two instance variables where *f* stands for some unspecified class. Operations can then be defined such as returning the last element in the list:

```
class list(f) is first: f; tail: list(f); end class;

procedure last(a: list(f)) returns f;
where f is free
begin
   loop
      if isempty(a.tail) then return(a.first);
      a := a.tail;
   end loop;
end last;
```

Saying *f is free* allows *f* to be replaced with any type when the procedure is used. Any time the class list is used, the parameter will be replaced with an actual class at compile time. The following program fragment uses the class list:

```
var
   a: list(int); b: list(real); i: int; r: real;
begin
   ...
   i := last(a); r := last(b);
```

Consider another example, sorting. A specification of a procedure to sort a list must state that there are comparison operators on the elements of a list. Using just *list(f)* where *f* is free only specifies that *f* is some class. One way to specify an additional operation is to use a procedure parameter:

```
procedure sort(a: list(f); greaterthan: proc(f,f) returns bool) returns
list(f);
where f is free
begin
```

```
    code for sorting
end sort;
```

This is a reasonable solution in this case. Often several different procedures are required to specify the partial behavior of a part. For example, some sort routines require greaterthan, lessthan, and equal. It quickly becomes tedious to pass all these procedures as parameters.

4 Descriptive classes

A descriptive class is used to avoid passing procedure parameters. Other classes can be instances of a descriptive class. A descriptive class specifies a partial behavior by specifying a set of procedure headings that describe what can be done with instances of classes that belong to the descriptive class. When a class is made an instance of a descriptive class, it automatically inherits all the operations on the descriptive class. A descriptive class to describe a total order would be:

```
descriptive class order(f) is
    eq: proc(order(f),order(f)) returns bool;
    grt: proc(order(f),order(f)) returns bool;
    less: proc(order(f),order(f)) returns bool;
end class;
```

To make the *int* class an instance of *order(f)* a declaration is used that gives actual procedures to pair with the headings:

```
instance of order(int) is equal, greaterthan, lessthan;
```

Procedures can be defined using descriptive classes:

```
procedure between(x:order(f); y:order(f); z:order(f)) returns bool;
where f is free
–returns true if x < y < z
begin
    return(x.less(x,y) and x.less(y,z));
end between;

procedure sort(a: list(order(f)) returns list(order(f));
where f is free
var
    m: list(order(f));
begin
    ... code to sort list
    return(m);
end sort;
```

When the type checking is performed, a class *C* is considered to be the same as a class *order(C)* if and only if *C* is an instance of the descriptive class *order*. A list of class *list(int)* can be sorted using the above procedure, since *list(order(int))* matchs *list(int)*.

The parameter on the class *order* is important. It makes calls like *between(3,"a",4)* illegal and a call like *between("3","a","4")* legal as long as the class *string* is an instance of *order*. The first example is illegal because the free type *f* in the procedure *between* is replaced by *int* in one place and *string* in another. This violates the constraint that a free type must be replaced with a single type within a call of a procedure. The second example is legal because *f* is replaced by *string* throughout.

Sometimes two parameters are useful on descriptive classes. For example, on a class *collection* similar to the Smalltalk class *Collection*, *e* will refer to the class of the elements and *c* to the class of the whole collection. The keywords **reference** and **name** refer to parameter passing modes:

```
class collection(c,e) is
   add: proc(e,c);
   remove: proc(e,c);
   iterate: proc(reference e, c, name statement);
end class;
```

The class *list(int)* could be made an instance of the class *collection* where *e* is replaced by *int* and *c* by *list(int)*. A class array could also be made an instance of *collection*. For type checking purpose *collection(c,e)* is consider the same as *c* if and only if *c* is an instance of *collection(c,e)*. For example, *collection(list(int),int)* would *match list(int)*.

5 Rapid Prototyping and Descriptive Classes

An example of a user interface with windowing will illustrate how descriptive classes are used for rapid prototyping. This example illustrates three techniques: How a descriptive class is used to describe a part; how a parameter is used to refine a part; and how a link is added from a subpart to the whole so that the part can work with the whole.

Each window is redrawn from a data structure instead of storing a bitmap. Each window is divided into panes. Each pane is described by a descriptive class pane:

```
descriptive class pane(f) is
   display: proc(pane(f));
   setplace: proc(pane(f),rectangle);
   controller: proc(pane(f),char,point);
end class
```

The *display* procedure tells how to display the pane, the *setplace* procedure tells where to display the pane, and the *controller* tells what to do when the user depresses a character with the cursor in the pane. The class *pane* captures the information the window manager needs about each window.

Parameters to classes are used to refine the behavior of existing classes. The existing class *titlepane(f)* implements a pane with one line of text. The parameter of *titlepane* is left unspecified. Provisions for a pop-up menu are provided. There is one instance variable of *titlepane* called *sub* that contains an instance of class *f*. This instance variable is used to refine the behavior of *titlepane*. For example, suppose we wanted a window that displayed

an integer and had a menu item that created a new window that displayed the next higher integer. The instance variable *sub* of the class *titlepane* will be used to keep the integer to be displayed. Two new procedures must be declared:

```
procedure menuitem(k:titlepane(int));
–describes the action associated with the menu item 'next'
var
   m: menu(titlepane(int)); t: titlepane(int); s: string;
begin
   m := newmenu();
   addmenuitem("next",menuitem,m);
   s := inttostring(k.sub+1) –string to be displayed
   t := newtitlepane(s,m);
   t.sub := k.sub+1;
   install(t); –tell the window manger about the new window
end menuitem;

procedure firstone returns titlepane(int);
–creates the first pane
var
   m: menu(titlepane(int)); t: titlepane(int)
begin
   m := newmenu();
   addmenuitem("next",menuitem,m);
   t := newtitlepane("0",m)
   t.sub := 0;
   return(t);
end firstone;
```

One common operation is to take two separate panes and merge them together into one pane. These panes could be merged one above the other or side by side. The relative size of the panes must also be specified. For this example assume windows are always merged one above the other with equal space given to each pane.

There is a class *merge(f,g)* for merging panes. If *f* and *g* are instances of the descriptive class *pane* then *merge(f,g)* is also. The following example merges a titlepane with the contents "start" above the first pane in the previous example.

```
procedure testmerge;
var
   m: menu(titlepane(bool));
   above: titlepane(bool);
   below: titlepane(int);
   combined: merge(titlepane(bool),titlepane(int));
begin
   m := newmenu();
   above := newtitlepane("start",m);
   below := firstone();
   combined := mergepanes(above,below);
```

```
    install(combined); –tell the window manager about the window
end testmerge;
```

When building a window out of parts, sometimes it is necessary for a subpane to know about a combination of other panes. For example, suppose that an menu item to redraw both panes was added to the top pane in the above example. The following change does this:

```
class whole is
    b: bool;
    w: merge(titlepane(whole),titlepane(int));
    – w is the link from the part to the whole
end class;

procedure menuitemredraw(k:titlepane(whole));
begin
    display(k.sub.w);
end menuitemredraw;

procedure testmerge2;
var
    m: menu(titlepane(whole))
    above: titlepane(whole); below: titlepane(int);
    combined: merge(titlepane(whole),titlepane(int));
    t: whole;
begin
    m := newmenu();
    addmenuitem("redraw",menuitemredraw,m);
    t := create;
    above := newtitlepane("start",m);
    above.sub := t;
    below := firstone();
    combined := mergepanes(above,below);
    t.w := combined
    install(combined); –tell the window manger about the window.
end testmerge;
```

Subclassing sometimes saves some space over refining a part with a parameter and avoids the need for a link from the subpart to the whole. This is because subclassing just adds more fields to the instances of the superclass. The whole and the part are the same object and so there is no need for the links.

6 Implementation

The basic idea behind the implementation of procedures that have descriptive classes as parameters is to add implicit parameters for each descriptive class used in the parameter list. These additional parameters are arrays of procedures that implement the operations

that describe the behavior of an object. For example, the *between* procedure that was defined earlier would have one additional parameter that was a vector of three procedures. When between is actually used, the free type is filled in with a specific type. Hence, the actual procedures that implement the comparison operators are known and a vector can be formed and passed to the procedure *between*. A better compiler could produce a procedure tailored to a particular instance of the class *order* instead of adding implicit parameters.

Forming the vector may require a considerable amount of work. For example, the *merge(f,g)* class is an instance of the class *pane* if and only if f and g are instances of *pane*. The actual procedure headings used to declare that *merge(f,g)* is an instance of *order* are:

```
procedure Mdisplay(merge(pane(f),pane(g)));
procedure Msetplace(merge(pane(f),pane(g)),rectangle);
procedure Mcontroller(merge(pane(f),pane(g)),char,point);
```

Each of these procedures has two implicit parameters to describe *pane(f)* and *pane(g)*. To form an array of procedures for *merge(f,g)*, these implicit parameters must be removed by forming new procedures that call the old procedures and fill in the arrays for *pane(f)* and *pane(g)*. These new procedures are then in the right form for forming the procedure array for *merge(f,g)*.

The major challenge of implementing descriptive classes is the type checking. A extension to the techniques found in [6] was used in the implementation of X2. There is not enough space here to give the details.

7 Discussion

Descriptive classes are better than subclassing in most cases. Descriptive classes have the advantages of compile-time type checking, yet retain or improve upon the important characteristics of subclassing. Like subclassing descriptive classes allow the sharing of code. One procedure defined on a descriptive class is shared among all class that are instances of that descriptive class.

Descriptive classes allow more possibilities for organizing classes. With descriptive classes, the class hierarchy is only used at compile time. Hence, it is possible to have different hierarchies for different scopes in a program. In Smalltalk, the hierarchy is used at run time which makes it difficult to have more than one global hierarchy.

Descriptive classes are better than abstract superclasses. The abstract behavior is more clearly documented with descriptive classes than with superclasses. An abstract superclass must be declared before any of its subclasses, but a descriptive class can be declared after the classes that are its instances.

A single class can be a member of many different descriptive classes. For example, The class *int*, which is already an instance of the descriptive class *order*, can be made an instance of the new descriptive class *ring* with:

```
descriptive class ring(f) is
   add: proc(ring(f),ring(f)) returns ring(f);
   mult: proc(ring(f),ring(f)) returns ring(f);
   sub: proc(ring(f),ring(f)) returns ring(f);
```

```
    zero: proc(ring(f))
end class;

procedure zero; begin return(0); end zero;

instance of ring(int) is addition, multiplication, subtraction, zero;
```

Since instances of descriptive classes do not inherit behavior from a descriptive class, no conflicts arise between behavior inherited from differ superclass as happens in multiple inheritance schemes.

Descriptive classes promise to be superior to subclassing. Descriptive classes provide the advantages or compile time type checking, yet retain the major advantages of subclassing.

8 References

[1] Bobrow, D. G., and Stefik, M. *The LOOPS Manual.* Xerox Corporation, 1983.

[2] Borning, A., and Ingalls, D. Multiple Inheritance in Smalltalk-80. *Proc. of the National Conference on Artificial Intelligence*, 1982. Pages 234-237.

[3] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[4] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. *CACM* 20(8), August 1977, 564-576.

[5] Sandberg, D. *The Design of the Programming Language X2.* Oregon State University. Technical Report 85-60-1,1985.

[6] Sandberg, D. *Type Checking and Parameterized Types.* Oregon State University. Technical Report 85-1-1,1985.

[7] Weinreb, D., and Moon, D. Objects, Message Passing, and Flavors. *Lisp Machine Manual,* 4th ed. Symbolics Inc., 1981.