Implementing mixins in Smalltalk

Terry Montlick

GRADY BOOCH DEFINES a mixin as:

> "A class that embodies a single, focused behavior, used to augment the behavior of some other class via inheritance; the behavior of a mixin is usually orthogonal to the behavior of the classes with which it is combined."[1]

There are always situations in which "mixing in" another class is the most straightforward thing to do. But mixins are generally thought of as requiring multiple inheritance,[2] which is the mechanism by which languages like CLOS supply them. Smalltalk, however, does not have multiple inheritance. Furthermore, multiple inheritance raises a host of issues, such as repeated inheritance—where the same superclass can be reached via more than route through the parent hierarchy.

Multiple inheritance raises a host of issues, such as repeated inheritance.

Instead, I'm going to take a simpler, more pragmatic approach. In this implementation, mixins are provided to a class by adding a message, mixins, which replies with a collection of mixin objects. If an object which uses mixins does not understand a particular message, then each object in the mixins collection is tried, in order. Technically, this is not true multiple inheritance because it provides only "interface inheritance" and not "class inheritance."

In order for this to work, the Object class must be modified.* Ordinarily, I am loath to do this, but this change is very simple and foolproof. Honest.

First, change the name of the existing doesNotUnderstand: method in the Object class to originalDoesNotUnderstand:. Then, add the following method:

```
doesNotUnderstand: aMessage
    "If the object has mixins, see if one responds to
     aMessage."
    (self class canUnderstand: #mixins) ifTrue: [
        self mixins do: [ :mixin |
            (mixin class canUnderstand: aMessage selector )
             ifTrue: [
                ^mixin perform: aMessage selector
                 withArguments:
aMessage arguments
                ].
            ].
        ].
    "was not handled by mixin, so pass to original
doesNotUnderstand handler"
                    ^self originalDoesNotUnderstand:
                     aMessage
```

The operation of this method is very simple. If the class of the object that generated the doesNotUnderstand: message selector understands the mixins message, then this message is sent to it. The reply is some kind of Collection, which is sent the do: message selector with a block argument. This block argument tests to see if an element of the Collection understands the message selector that the original object did not understand. As soon as such an element is found, it is sent the message selector, along with the original arguments.

The mixins may be put in any type of collection. A simple OrderedCollection may be used as the mixins object. A dictionary might also be used to gain named access to the individual mixin objects.

Here is a simply and highly artificial, but illustrative, example of using mixins. A class, MixinTest, has a single instance variable called mixins. It has the following accessor:

```
mixins
    ^mixins
```

and the following initialize method:

```
initialize
    mixins := OrderedCollection new.
    mixins add: (Date today).
```

* You could alternatively create a class that overrides the doesNotUnderstand: message selector of the Object class, and always subclass from this. However, this restricts the utility of mixins, since you would not then be able to "mixin" to an existing class hierarchy.

The initialize method sets the mixins instance variable as an OrderedCollection with a single element, a Date, which is today's. This initialize method is sent by the instance creation method:

```
new
    ^super new initialize
```

Inspecting the following statement causes the current day of the month to be displayed:

```
MixinTest new dayOfMonth
```

Additional levels of mixins can be added. Consider a class Mixin2Test, which is a subclass of MixinTest. This class also has the instance creation method:

```
new
    ^super new initialize
```

but the initialize method is

```
initialize
    super initialize.
    mixins add: (Point x: 100 y: 50)
```

By calling super initialize, the MixinTest object creates the mixins instance variable and adds a Date object to it. The Mixin2Test initialize method then adds a Point object to mixins.

A Mixin2Test object now understands messages for two mixin objects. For example:

```
Mixin2Test new dayOfMonth
Mixin2Test new x
```

An object of any class may act as a mixin. However, an abstract mixin class is particularly powerful. An abstract mixin is a class that cannot be instantiated in isolation, because it requires another object or objects to provide methods for it.

I'll call the object that a mixin is added to the root object. A class for subclassing abstract mixin classes is called AbstractMixin. It has a single instance variable called root, which has the accessors:

```
root
    ^root

root: anObject
    root := anObject
```

It also has the class-side instance creation method:

```
root: anObject
    ^self new root: anObject
```

Subclasses of AbstractMixin must declare their root object when they create a new instance. The root object is implicitly referred to by sending messages of the form:

```
self root <some message for root object>
```

Every AbstractMixin subclass should clearly document what methods it requires its root object to provide.

As an example of abstract mixins, I'll use one given by Seidewitz.[3] The abstract mixin class InterestMixin is a subclass of AbstractMixin and provides the incremental functionality of earning interest. It has a single-instance, variable, rate, with the usual accessors, plus a method for computing interest earned:

```
interestEarned: dt
    "root must provide a 'balance' method"
    ^root balance * rate * dt
```

An instance creation method sets both the rate and the root object:

```
rate: aNumber root: anObject
    ^(self root: anObject) rate: aNumber
```

Another mixin class, this time a concrete mixin called AccountMixin with an instance variable called balance, provides basic account functionality:

```
deposit: aNumber
    balance := balance + aNumber

withdraw: aNumber
    balance := balance - aNumber
```

An instance creation method sets the current balance:

```
balance: aNumber
    ^(self new) balance: aNumber
```

Now, I'll put these together in a new class called SavingsAccountImplementation, which has an instance variable mixins with an accessor of the same name. All that is necessary is add the instance initialization method:

```
balance: aBalance rate: aRate
    mixins := OrderedCollection new.
    mixins add: (AccountMixin balance: aBalance).
    mixins add: (InterestMixin rate: aRate root: self).
```

and the following class-side instance creation method,

```
balance: aBalance rate: aRate
    ^self new balance: aBalance rate: aRate
```

That's all there is to it! The AccountMixin object provides the implementation of the balance method, which is required by the InterestMixin. SavingsAccountImplementation does not have to supply anything.

References

1. Booch, G. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Benjamin/Cummings, Menlo Park, CA, 1994, p. 515.
2. Gamma, E., *et al. Design Patterns.* Addison-Wesley, Reading, MA, 1995, p 16.
3. Seidewitz, E. "Controlling Inheritance," *Journal of Object-Oriented Programming* 8(8), Jan. 1996.

Terry Montlick is the founder of Software Design Consultants, which specializes in state-of-the-art Smalltalk projects. He can be reached by email at 75260.2606@compuserve.com or at http://www.softdesign.com/softinfo/sdc.html.