Kent Beck

Object-Oriented Recursion

Understanding recursion is a watershed in the life of most software developers. The idea that you define a computation, not in terms of other computations, but in terms of itself, is a mind bender for most people. I can remember carefully drawing stack frames with their own local storage and program counter and painstakingly following the progress of factorial and depth-first binary tree traversal. It was only when I found an obscure little book in the science library that explained how to transform recursion into iteration and vice versa, that I really felt I understood recursion. Even then, it was months before I could reliably use it as a programming technique.

Such a powerful technique must be an important part of programming objects, right? Well, yes and no. Combining recursion with objects is powerful, more powerful than its procedural counterpart, but you have to manage it differently to make effective use of it.

TAKE ONE

Rule 1: Send the recursive message to different objects.

Procedural recursion is defined as a procedure that calls itself with different parameters. At some point, you have to reach the base or degenerate case of the recursion, at which time you do not call the procedure further (not if you want the program to terminate, anyway). Factorial implemented with procedural-style recursion looks like this:

```
Object>>factorial: aNumber
    ^aNumber = 1
        ifTrue: [1]
        ifFalse: [aNumber * (self factorial: aNumber - 1)]
```

In this version of #factorial: the receiver of the message plays no particular role. The existence of the receiver of a message as the implicit first parameter motivates the first change in the use of recursion with objects. Rather than invoke what is in essence a subroutine over and over on the same object with different parameters, object-oriented recursion invokes the same routine, but with different objects as the receiver. The object-oriented version of factorial doesn't need an additional parameter. The receiver of the message is the number to be "factorialed."

```
Number>>factorial
    ^self = 1
        ifTrue: [self]
        ifFalse: [self * (self - 1) factorial]
```

The resulting code is simpler by one argument, but otherwise looks much like the procedural version.

WE PAUSE FOR A BIT OF MATHEMATICS

To illustrate the other difference between procedural and object-oriented styles of recursion, we will turn to Peano's Axioms of Arithmetic. Zero is represented as "zero." Other positive numbers are defined as nested invocations of the function "succ" (for "successor"). For example, three is represented as:

succ(succ(succ(zero)))

Given this definition of numbers, we can now define addition recursively. The base case of the recursion is adding any number to zero, equals that number:

Case 1: add(X, zero) = X

Thus, adding zero and three results in three:

add(succ(succ(succ(zero))), zero) = succ(succ(succ(zero)))

The recursive case of the definition says that adding X to a number which is the successor of Y is the same as adding X to Y, then getting the successor of the sum.

Case 2: add(X, succ(Y)) = succ(add(X, Y))

Algebraically, this is the same as saying:

X + (1 + Y) = 1 + (X + Y)

Adding two to one results in the following invocations:

add(succ(zero), succ(succ(zero))) = succ(add(succ(zero), succ(zero))) by case 2
succ(add(succ(zero), succ(zero))) = succ(succ(add(succ(zero), zero))) by case 2
succ(succ(add(succ(zero), zero))) = succ(succ(succ(zero))) by case 1

Lo and behold, 2 + 1 = 3!

AXIOMS TO OBJECTS

We can turn Peano's Axioms into objects by making a successor object, which is linked to its predecessor. A

linked list of three successors represents the number three. The end of the list will be represented by nil.

```
Class: Succ
    superclass: Object
    instance variables: pred
```

We can provide a Constructor Method for Succ that returns the Peano version of an Integer:

```
Succ class>>fromInteger: anInteger
    ^anInteger = 0
        ifTrue: [nil]
        ifFalse: [self of: (self fromInteger: anInteger - 1)]
```

We create the successor of a Peano number by creating a new instance of Succ and setting its predecessor to the number.

```
Succ class>>of: aPeanoNumber
    ^self new setPred:aPeanoNumber
Succ>>setPred: aPeanoNumber
    pred := aPeanoNumber
```

We compute the predecessor of a Peano number by simply returning the value of the instance variable "pred."

```
pred
    ^pred
```

We compute the successor by tacking on another successor object:

```
succ
    ^Succ of: self
```

For debugging purposes, we can define a printing method that shows us the receiver in Peano format.

```
Succ>> printOn: aStream
    aStream nextPutAll: 'succ('.
    self pred isNil
        ifTrue: [aStream nextPutAll: 'zero']
        ifFalse: [self pred printOn: aStream].
    aStream nextPutAll: ')'
```

Three now prints as three nested invocations of "succ":

```
(Succ fromIneger: 2)+ (Succ fromInteger: 1) succ(succ(succ(zero)))
```

Given the definitions of #pred and #succ, we can turn the axioms of arithmetic into a method. Because we are explicitly checking for the base case of the recursion, the code is not quite a direct translation of the original axioms.

```
+ aPeanoNumber
    |subTotal|
    subTotal:= self pred isNil
        ifTrue: [aPeanoNumber]
        ifFalse: [self  pred+ aPeanoNumber].
    ^ subTotal succ
```

Adding two and one result in our now famous three:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(zero)))
```

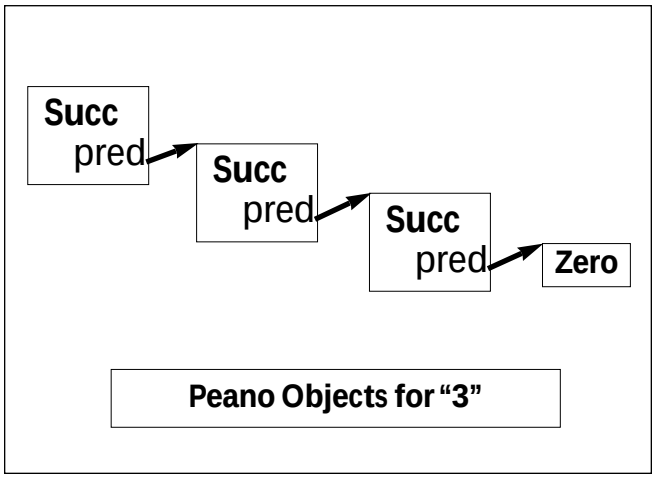

Figure 1. The number three, represented as objects.

Take Two

Rule 2: Represent the base case of the recursion by a distinct object. Now we are finally ready to examine the second difference between procedural and object-oriented recursion. Procedural recursion relies on explicits checks for the base case of the recursion. The previous code shows this style in the #+method, where an explicit conditional checks for a nil argument.

Every time I use recursion in the beginning, and distressingly often thereafter, I forget to check the base case. You can use objects and messages to make such errors less likely, and to simplify the code at the same time.

The key is not to rely on the generic undefined object to stop the recursion. Instead, you create your own "undefined object," then make sure it responds to the same messages as the object representing the recursive case.

To apply this principle here, we have to first replace nil with a new object, Zero.

```
Class: Zero
    superclass: Object
    instance variables: <none>
```

Rather than returning a nil when we want to represent a zero, we return an instance of our new object instead:

```
Succ class>>fromInteger:anInteger
    ^anInteger=0
    ifTrue: [Zero new]
    ifFalse: [self of: (self fromInteger: anInteger - 1)]
```

Adding a Zero and any number results in that number:

```
Zero>>+ aPeanoNumber
    ^aPeanoNumber
```

Adding a successor to a number now need not check for nil:

```
Succ>>+ aPeanoNumber
    ^(self pred + aPeanoNumber) succ
```

Once again, we can add two and one to get three:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(a Zero)))
```

Notice that the zero prints out a little differently than before. We can easily fix that:

```
Zero>>printOn: aStream
    aStream nextPutAll: 'zero'
```

And we can get rid of the code in Succ that checks for nil. I love bug fixes that involve removing code!

```
Succ>>printOn: aStream
    aStream  nextPutAll: 'succ('
    self pred printOn: aStream.
    aSteam nextPutAll: ')'
```

Now two plus one prints correctly again:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(zero)))
```

The addition code looks much more like the original mathematics (taking postfix notation into account):

```
Case 1: add(zero, X) = X
Case 2: add(succ(X), Y) = Y = succ(add(X, Y))


Zero>>+ aPeanoNumber
    ^aPeanoNumber
Succ>>+ aPeanoNumber
    ^(self pred + aPeanoNumber) succ
```

We have been able to use polymorphism to write code that communicates more clearly, because it translates more directly from the original source. The code is more like a specification and less like a computer program.

The Two Ways

We have seen two ways in which object-oriented recursion differs from procedural recursion. First, rather than invoke the same procedure with different arguments, object-oriented recursion represents the invocations themselves as objects, sending the same message to different objects.

In our example, this corresponded to creating a new object to represent one invocation of the successor function. Most recursive routines don't require this (somewhat unnatural) step. If the recursive routine is operating over a recursive data structure (trees or lists, for example), the objects are likely to be there already.

The second difference between procedural and

"object-oriented recursion represents the invocations themselves as objects, sending the same message to different objects."

object-oriented recursion is in the use of a special-purpose object to represent the base case of the recursion. Polymorphism's ability to capture decision making in what would otherwise be a simple procedure call comes to the fore in this technique. The resulting code communicates, clearly even in the absence of explicity conditionals.

You might ask, "Why don't you use Smalltalk's built-in special object, nil, to represent the base case of the recursion?" After all, in the example above, we could implement #+ in UndefinedObject just as we did in Zero and the code would work fine. The problem is that all developers share the same UndefinedObject. If everyone added a handful of methods to it, the result would be thousands of methods on UndefinedObject, in other words, chaos. The chances of such code communicating clearly, are slim, even if there weren't accidental disagreements about what UndefinedObject>>+ should do.

If you'd like to play around with recursion, you may want to extend the code above. Try implementing #- or #*. I found implementing negative numbers (hint, you need a Pred class) to be quite challenging. **S**

Kent Beck has been discovering Smalltalk idioms for twelve years at Tektronix, Apple Computer, and MasPar Computer. He is the founder of First Class Software, which develops and distributes developer tools for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, 408.338.4649 (voice), 408.338.3666 (fax), or by email at 70761,1216 (Compuserve).