JustCloningAround *subclass:* #CloningExtensions

Keith Piraino

All applications wrestle with the problem of copying objects at some point. In the course of some recent work, I used a variation of the cloning approach published in the *Journal of Object-Oriented Programming* (JOOP) two years ago.[1] This article will describe two extensions I added:

- Allow for customized behavior after the deep copy (#postDeepCopy:); and
- Provide ability to NOT copy certain instance variables.

We need a class method that will traverse the superclass chain and collect the names of all instance variables to not copy.

JUSTCLONINGAROUND

The JOOP article described an implementation of a deep copy that can handle arbitrarily deep object structures in all three major Smalltalk dialects. Circularities are handled by keeping track of the objects copied in an IdentityDictionary. The keys are the original object and the values are the copies.

As the title suggests, I look at this article as a "subclass" of the JOOP article. Just as you wouldn't expect to understand a class without browsing its superclass, you shouldn't expect to understand this article without having read the JOOP article. I'll present a complete implementation in code,[*] but I won't cover the concepts from the original article.[†]

EXAMPLE

Consider the OMT[2] style diagram in Figure 1. Each customer contains a collection of invoices. Each invoice contains a collection of ordered items. These are aggregation or ownership[3] relationships.

Each invoice holds a reference to the customer as well as to the salesperson that created the invoice. Each ordered item holds a reference to the product it is ordering. These are not aggregation relationships. For example, an invoice does not own the salesperson.[4]

Consider the instance diagram in the top half of Figure 2.[‡] Jane Profit might use this existing invoice as the basis for creating a new one. The bottom half of Figure 2 shows the resulting instance diagram, if the JOOP approach had been used to copy the invoice. There are two problems with this result. The first is that everything is copied when what we want is to copy only the objects referenced by aggregation relationships. For example, what we don't, want is a new instance of Jane Profit.

The second problem is that when we copy an invoice we want to reset the quote to the customer. The quote is what the customer actually pays, and it might be higher or lower than the actual cost. Determining the quote is up to the salesperson. In this particular case the customer has been given a $25 discount on an order for three "blue widgets;" however, that doesn't mean he or she will always get that discount. In this application we've established the rule that all quotes should be set to zero when copies are made.

After copying, what we actually want is shown in Figure 3.

[*] I used VisualWorks, but this should work in other dialects if you keep in mind the issues raised in the JOOP article.

[†] The *JOOP* article also describes how to implement a deep equal, which I don't cover at all.

[‡] I'm not showing the Customer to Invoice relationship for now.

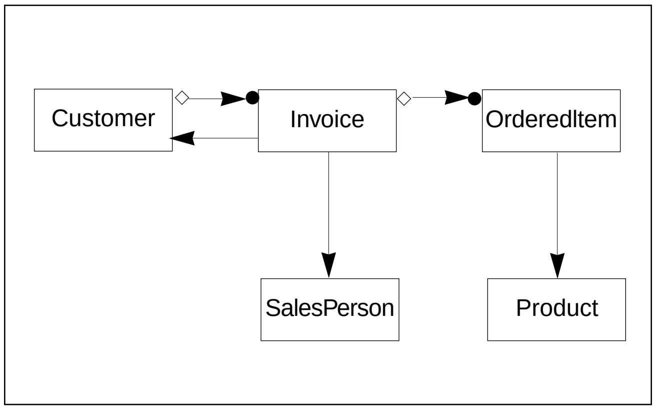

Figure 1. Example Object Model.

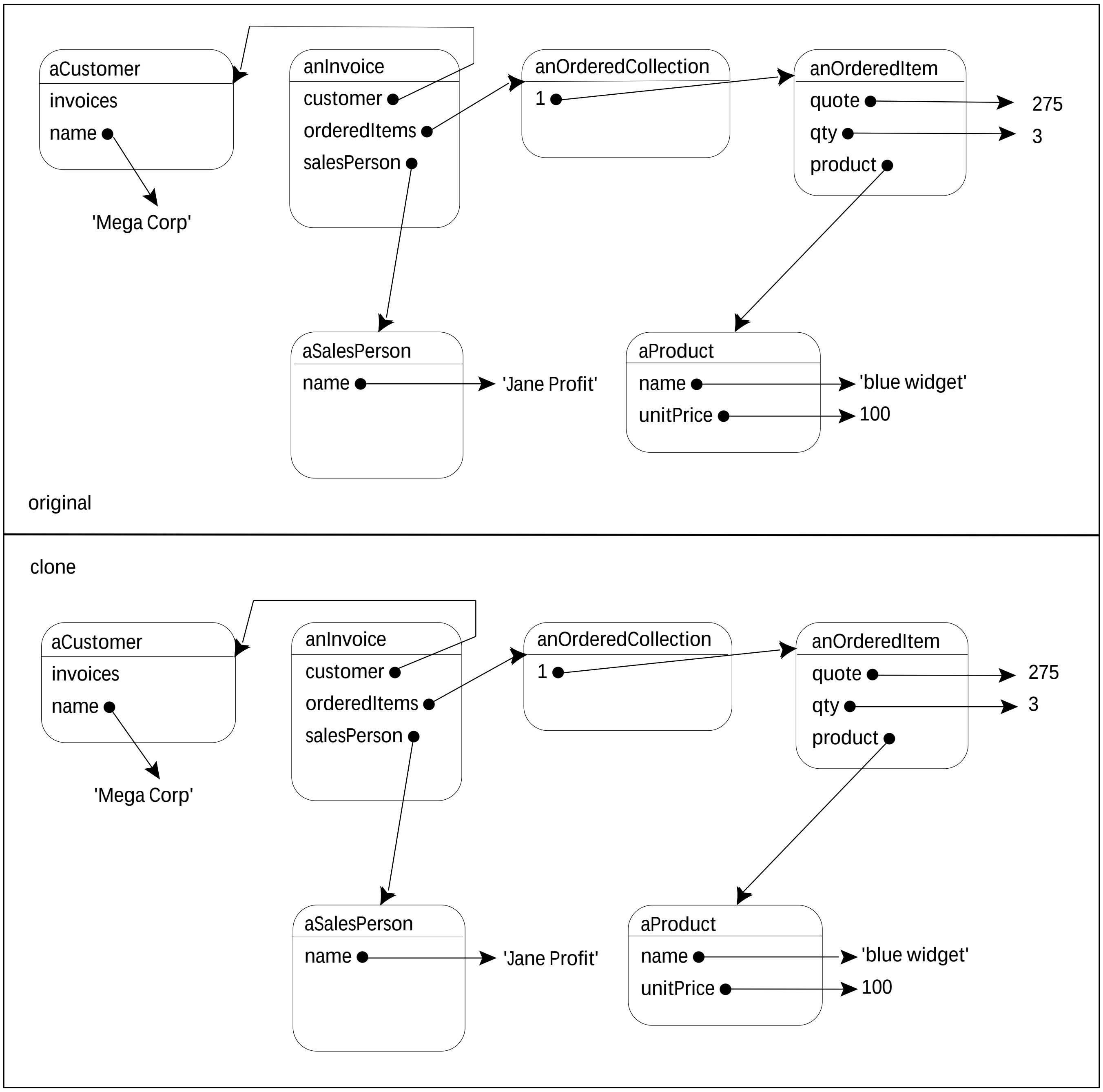

Figure 2. Cloning an invoice using the JOOP approach.

#POSTDEEPCOPY:

Of our two problems, resetting the quote is the easier one so let's tackle that first. Our generic problem is being able to have customized behavior after an instance is copied. Using the normal VisualWorks mechanisms, this would be done via #postCopy. The equivalent to this in our generic mechanism is a method called #postDeepCopy: that is sent to the new instance after the deep copy is done. Listing 1 shows the implementation of the generic copy mechanism. It's been factored a little differently, but other than #postDeepCopy: this is the same code as published in the JOOP article.

Below would be the implementation of this method for OrderedItem:

```
postDeepCopy: anObject
    super postDeepCopy: anObject.
    self quote: 0.
```

The anObject parameter is the original object. I've had no cause to use it yet, but I thought this parameter might come in handy in certain cases. For instance, we could use #postDeepCopy: as a way to solve the problem with copying objects that aren't owned. After the copy is made, we simply set the appropriate instance variable to refer back to the original object. Below is an example for Invoice:

```
postDeepCopy: anObject
    super postDeepCopy: anObject.
    self salesPerson: anObject salesPerson
```

One problem with this approach is that there could be side effects to copying a salesperson object, which we want to avoid. Let's see if we can come up with a better approach.

#DONTCOPYVARS

For any given object we want only to copy some of its instance variables. We have a choice, though. We can create a mechanism that forces us to specify what should be copied, or that requires us to specify what should NOT be copied. I've taken the latter approach because, in problem domains I've been exposed to, it appears that more things get copied than not.

Ideally, the vendors should standardize on some way of specifying "ownership" and make use of it in their copy mechanisms.

This mechanism will also assume that we can specify this information on a class basis. In other words, no instance of Invoice will ever need to create a new instance of SalesPerson when it is copied.

Every class can optionally define a class method called #dontCopyVars that answers a collection of the named instance variables that should not be copied. Below are examples from our object model in Figure 1:

```
Invoice class>>dontCopyVars
    ^#(#customer #salesPerson)

OrderedItem class>>dontCopyVars
    ^#(#product)
```

Customer, SalesPerson, and Product would not have to define a #dontCopyVars method.

#ALLDONTCOPYVARS

In our example we don't have to worry about inheritance, but most structures aren't this simple. We need a class method that will traverse the superclass chain and collect the names of all instance variables to not copy. The code below is patterned after #allInstVarNames§ and #accumulateInstVarNames but, takes into account that not every class will implement #dontCopyVars:

```
Object class>>allDontCopyVars
    | vars |
    vars := OrderedCollection new.
    self accumulateDontCopyVars: vars.
    ^vars

Object class>>accumulateDontCopyVars: aCollection
    self superclass notNil
        ifTrue: [self superclass accumulateDontCopyVars:
aCollection].
    (self class includesSelector: #dontCopyVars)
    ifTrue: [aCollection addAll: self dontCopyVars].
```

§ You can find a lot of good solutions if you try to think of how your problems are similar to something already in the image.

These two methods ensure that every class can answer a collection of the instance variables that should not be copied. All that's left is a little bit of code to utilize this information during copying. See Listing 2.

MISCELLANEOUS

Every object should answer a copy of itself when sent the message #copy. Users of the object should not have to worry about whether it uses the deep copy mechanism. To account for this, I usually redefine #copy for all my domain objects as follows:

```
copy
    ^self deepCopy
```

When we copy an invoice we need to know whether we are making a copy for use with the customer that owns the original invoice, or for a different customer. To handle this I defined a #copyFor: method in Invoice that takes the Customer as a parameter:

```
copyFor: aCustomer
    | aCopy |
    aCopy := self copy.
    aCustomer invoices add: aCopy.
    aCopy customer: aCustomer.
    ^aCopy.
```

Avoid the temptation to specify #dontCopyVars based on the current functionality of your application. As an example, assume that OrderedItem instances have a reference to their containing Invoice. Also assume that the application currently allows Invoices to be copied, but not OrderedItems. You might be tempted to not include #invoice in #dontCopyVars for OrderedItem. Because of the support for circularities in the copy mechanism, if Invoice is copied before OrderedItem the OrderedItem will end up pointing to the correct Invoice anyway.

However, at some point these kinds of assumptions will come back to haunt you. Every object should be able to answer a reasonable copy of itself.

CONCLUSION

Unfortunately, Smalltalk does not provide a way to distinguish between aggregation relationships and simple references. What I've provided is one way of doing this. Ideally, the vendors should standardize on some way of specifying "ownership" and make use of it in their copy mechanisms. This kind of standard would, for example, allow CASE vendors to output this information during code generation.

LISTING 1

instance methods in Object

```
deepCopy
```

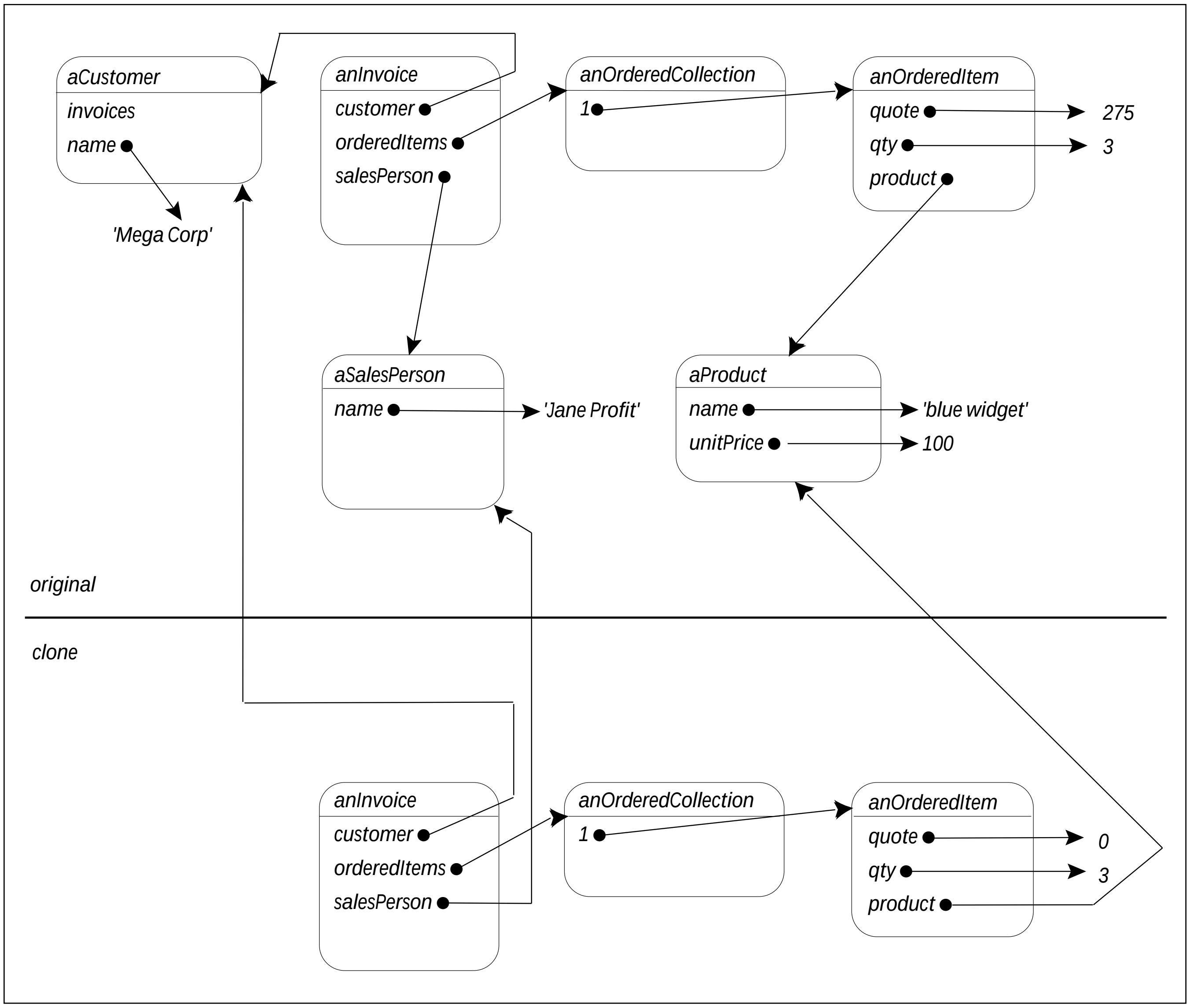

Figure 3. Cloning an invoice using both extensions.

```
    ^self deepCopyWithoutRecopying: IdentityDictionary new.

deepCopyWithoutRecopying: anIdentityDictionary
    ^anIdentityDictionary
        at: self
        ifAbsent: [self doDeepCopyWithoutRecopying:
anIdentityDictionary]

doDeepCopyWithoutRecopying: anIdentityDictionary
    | aCopy |
    aCopy := self shallowCopy.
    aCopy == self ifTrue: [^self].
    anIdentityDictionary at: self put: aCopy.
    self class isPointers ifFalse: [^aCopy].
    aCopy releaseCopyDependents.
    aCopy copyNamedVarsWithoutRecopying:
anIdentityDictionary.
    aCopy copyUnnamedVarsWithoutRecopying:
anIdentityDictionary.
    aCopy postDeepCopy: self.
    ^aCopy.

releaseCopyDependents
    ^self breakDependents

copyNamedVarsWithoutRecopying: anIdentityDictionary
    | newPart |
    1 to: self class instSize do:
        [:idx |
        newPart := (self instVarAt: idx)
        deepCopyWithoutRecopying: anIdentityDictionary.
    self instVarAt: idx put: newPart].

copyUnnamedVarsWithoutRecopying:
anIdentityDictionary
    | newPart |
    1 to: self basicSize do:
        [:idx |
        newPart := (self basicAt: idx)
            deepCopyWithoutRecopying:
```

```
anIdentityDictionary.
        self basicAt: idx put: newPart].

postDeepCopy: anObject
    "The receiver is a deeply copied instance of anObject.
    Subclasses may override this method to provide
behavior after copy is done"
```

LISTING 2

instance methods in Object

```
copyNamedVarsWithoutRecopying: anIdentityDictionary
    | newPart |
    self class allVarIndicesToCopy do:
        [:idx |
        newPart := (self instVarAt: idx)

            deepCopyWithoutRecopying:
anIdentityDictionary.
    self instVarAt: idx put: newPart].
```

class methods in Object

```
allVarIndicesToCopy
    "Answer a collection of the indices of all named
variables to copy for the receiver"
    ^self allVarNamesToCopy collect:
    [:each | self allInstVarNames indexOf: each]

allVarNamesToCopy
    "Answer a collection of all variable names to copy for
the receiver"
    ^self allInstVarNames reject:
    [:each | self allDontCopyVars includes: (each asSymbol)]

dontCopyVars
    "Answer a collection of instance variables defined in
this class that should not be copied when a deep copy is
made of an instance of the receiver or one of its
subclasses. The collections should contain symbols, not
strings.

Only classes that define instance variables that shouldn't
be copied need to define this method"
    ^#()
```

REFERENCES

1. Lalonde, W. and Pugh, J., "Just Cloning Around," *JOOP*, 7(5); 1994.
2. Rumbaugh, J. et al., OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
3. Weir, C., "Improve Your Sense of Ownership: Exploring a Design Principle," *ROAD*, 2(6); 1996.
4. Check out www.sigs.com/publications/docs/oc/9608/oc9608.d.dialog.html for an interesting discussion of aggregation vs. association.

Keith Piraino is a consultant who can be reached at keith.piraino@bug.com.