Proper use of class methods

Jill Nicola

The biggest blunder most new Smalltalk developers make is improper use of class methods. Many a new Smalltalk convert has implemented major portions of a system design using class methods. Well, why not? They are easy to access—callable from anywhere by any object.

Tempting, but not a good idea. First, consider your system design documents. There is no place on an object model for class methods. When you build an object model of your system design, the services in a class symbol map to instance methods, not class methods. For example, consider the simple class symbol for a product vendor as shown in Fig. 1:

Every product vendor has its own name, address, and list of products offered. These attributes that every product vendor possesses translate into Smalltalk instance variables defined in the ProductVendor class. Every product vendor can be asked to provide a quote for a given quantity of product. This is a behavior every product vendor provides, so it translates into an instance method defined in the ProductVendor class.

Now at some time during system execution all the product vendor objects must be loaded into memory. One approach is to write a class method, loadAll, in the ProductVendor class. This class method goes out to the database and reads all the product vendor objects. A class method is used so it can be called by a human interface screen or some other process. Great, but where does it go on the object model? To show it on the class symbol would imply it is an action every product vendor performs—an instance method (see Fig. 2).

Faced with this problem, most developers invent an extension to the notation or discover within the notation some obscure demarcation for distinguishing class methods from instance methods. Oh boy. If you follow this approach, you have an object model with notation only you understand that is implemented using global functions. That's right, global functions.

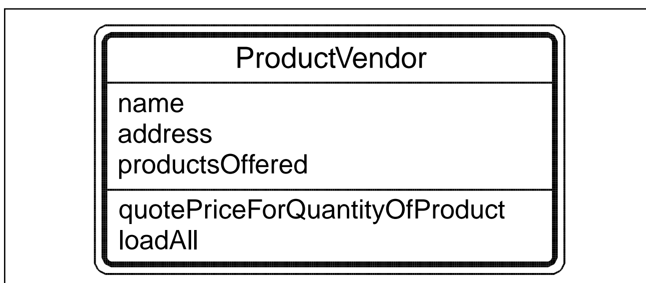


Figure 1. Simple class symbol.

Your design is much less object oriented with class methods. Ideally, you want *all* the functionality in your system to be implemented by *objects* that represent things in the real world, or correspond to system components. Class methods are a fluke, a side effect of the fact that most object-oriented languages need a data structure to serve as the definition for producing objects, and this data structure, typically called a class, needs to have behavior so it can produce objects. Making use of this fluke is just an acknowledgment that you have system functionality you cannot associate with any object in your design . . . global functionality.

Another problem arises when requirements change. Suppose you want to use the ProductVendor class in two different applications—one with an ORACLE database, one with a OODBMS. The loadAll method would have to be written differently for each application. In fact, every time the data representation requirements changed you would need a new version of the loadAll method. Ouch.

So to recap, the problems with class methods are: 1) Class methods require new or obscure notation on the object model. 2) Class methods make your design less object oriented. 3) Class methods decrease the flexibility of your system to changes in requirements.

What to do? What to do? First, learn to recognize that wherever class methods abound . . . objects are missing. In the example, a loadAll class method was used because there was no object in the system design responsible for loading product vendors. So add one (see Fig. 3).

Heck, add a bunch… ProductVendorOracleDBA, ProductVendorOODMBSDBA. However many you might need, just get that loading behavior out of the class method on your business domain object and get it into an instance method on a separate object. Now in all your future applications whenever product vendors need to be loaded, create a product vendor DBA of the proper class, hook it to the server, and tell it to loadAll. Gee, you could even document the loading procedure with an object interaction or a scenario diagram, something that would be really hard if you were using class methods.

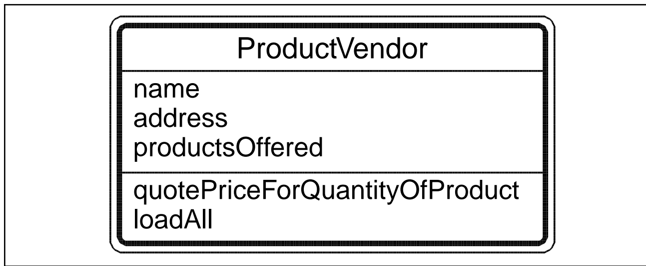


Figure 2. An Instance method.

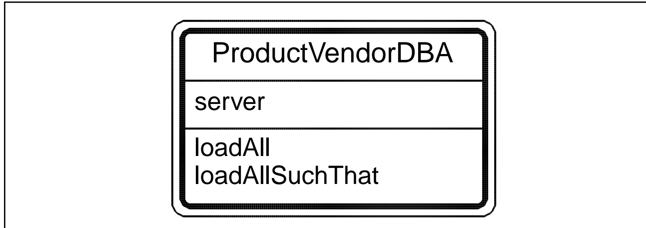


Figure 3. Object for loading product vendors.

So, system functionality in class methods, particularly in a business domain class, indicates objects are missing. Here's when you should use class methods:

1. *Creating new objects.*

```
Product new
Date today
Time now
perot
Array new: 10
```

Messages new, today, now, perot, new:, correspond to class methods that create new objects. The new class methods typically create objects with nil or default initial values in the instance variables. The other methods, such as today, and now, create special objects in the class that have instance variables preset to meaningful values.

2. *Managing class variables.* Use class methods to:
   - Initialize, reset, flush class variables.
   - Provide read and write access into class variables.

Wow. That's a short list. Well, there is one other time you might consider class methods, but it is for development purposes not system design.

3. *Creating example or test objects.*

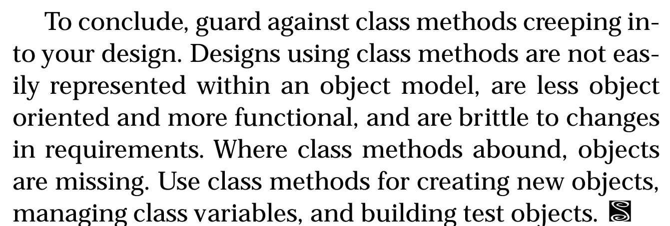
A test object has in its instance variables typical data values that would exist during a normal system execution. Test objects are a great help during development because they allow developers to run portions of the system without having to load data, guess at representative data values, or keep workspaces open with scripts for building objects. Here's how a typical test object method looks. Note, how the executable comment within the method makes it easy to run.

```
ProductVendor class methodsFor: 'examples'

testObject
   "ProductVendor testObject"

| vendor |
vendor := self new.
vendor name: 'Vendor X'.
vendor address: Address testObject.
vendor addProductOffered: Product testObject.
vendor addProductOffered: Product testObject2.
vendor addProductOffered: Product testObject3.
^vendor
```

Now do not go off and implement elaborate test scenarios with a slew of class methods; you will be making the *missing object* mistake all over again. Test scenarios will vary from application to application, so build separate objects to implement your testing procedures. Test objects are essentially the unit tests from which all test procedures are built.



To conclude, guard against class methods creeping into your design. Designs using class methods are not easily represented within an object model, are less object oriented and more functional, and are brittle to changes in requirements. Where class methods abound, objects are missing. Use class methods for creating new objects, managing class variables, and building test objects.

Jill Nicola is President of JEN Consulting, which offers Smalltalk training, consulting, and mentoring services. She specializes in architecture design and customized GUIs. She can be reached by email at nicola@jencon.com.