are dealing with (it could be a dictionary, an array, a list...). Method add: is to be implemented in the subclasses:

```
add: newObject
    "Include newObject as one of the receiver's elements. Answer newObject.
    ArrayedCollections cannot respond to this message."

    self subclassResponsibility
```

This "abstract" definition of add: even allows us to define in the Collection methods that use it, such as add:withOccurrences:, which is:

```
add: newObject withOccurrences: anInteger
    "Add newObject anInteger times to the receiver. Answer newObject."

    anInteger timesRepeat: [self add: newObject].
    ^ newObject
```

We can even do without defining add: at all; add:withOccurrences: would still be defined as just shown, and Smallktalk will not balk as long as at runtime the receiving object has add: defined. (By the way, add:WithOccurrences: is a nice little implementation of the Strategy pattern.) At the same time, the comment in add: points out that some subclasses of Collection, those rooted at its ArrayedCollection subclass, should not implement the message at all. This, again, is enforced only at runtime, by using shouldNotImplement:

```
add: newObject
    self shouldNotImplement
```

There is nothing inherently wrong in using conventions in programming; part of the art is mastering conventions. What can be problematic, however, is depending solely on collections to obtain the required result. Smallktalk will not warn us if we forget not to implement add: in ArrayedCollection. We will only fail miserably at runtime.

We saw earlier how easy it is to implement a proxy class in Smalltalk. The truth is, though, that if we actually want a proxy class that stands as a proxy for only a small number of methods, things get more complicated. The reason has to do with the absence of real abstract classes. A proxy class may be a subclass of the class we want to proxy, in which case it will inherit all the methods of the proxied class, and not just those methods that we want to proxy. Alternatively, we may employ latent typing and define in the proxy class only the proxied methods; the problem is that, since everything is an object, the proxy class will inherit all the methods of the Object class. Ideally, we would like a class that proxies two methods to have these two methods only, but it is not obvious how we can achieve that. The class will have all the methods inherited by its ancestors. We can resort to tricks to minimize the number of inherited methods; for example, in some Smalltalk dialects it is possible to make a class a subclass of nil instead of Object. In this way nothing is inherited, but we need to copy and paste some necessary methods from Object (Alpert et al. 1998, p. 215). In Squeak, we can make it a subclass of ProtoObject instead of Object.