```
class Bird {
public:
    virtual void fly();   // birds can fly
    // ...
};

class Penguin: public Bird {    // penguins are birds
public:
    virtual void fly() { error("Attempt to make a penguin fly!"); }
    // ...
};
```

C++ programmers may alternatively hide the offending method:

```
class Base {
public:
    virtual void f()=0;
};

class Derived: public Base {
private:
    virtual void f() {
    }
};
```

So, `Derived` is no longer an abstract class, but it still does not have an `f()` function that can be of any use. Such shenanigans are to be avoided.

In Java, you can get away with it again by returning an error or throwing an exception; you can also making the nonsensical method abstract in the subclass, thus making the class hierarchy abstract from that point downward until you remake the method concrete. Again, such shenanigans are to be avoided.

The usual way to design around the problem of inapplicable or irrelevant methods in a class hierarchy is to redesign that hierarchy. A class hierarchy that would better reflect the peculiarities of the avian world would introduce a `FlyingBird` subclass of Birds, for those birds who do fly, and make `Penguin` a direct subclass of `Bird`, and not of a `FlyingBird`.

In Squeak we find a mere 45 methods that send the `shouldNotImplement` message, which is used when a method inherited by a superclass is not applicable in the current class. This is a very small proportion of the total number of methods and objects in Smalltalk, so the language is not fraught with badly designed class hierarchies. However, even the `shouldNotImplement` message is actually an implementation. This hints at a deeper issue in Smalltalk, which is that we do not have real abstract classes or methods. Methods are abstract by convention; there are no methods that have no implementation at all.

Instead of using the `shouldNotImplement` message, we could specify that a given method is the responsibility of a subclass, which we have seen is what the `subclassResponsibility` message is for. The class from which we send `subclassResponsibility` is then by convention abstract. For instance, the `Collection` class gives a generic interface for adding and removing objects, but does not provide implementation, as the implementation varies depending on the subclass we

Download at WoweBook.Com