When we start looking into what really gets inherited from where, things become intricate. As everything in Smalltalk is an object, including classes, we may wonder what kind of instances classes are. It turns out that classes are instances of metaclasses; a class's metaclass is named after the class name with the word "class" after it. For example, the metaclass of `Object` is `Object class`. This is a sensible convention, but it does not really answer the question, as we may wonder what kind of instances metaclasses are. It turns out that metaclasses are instances of class `Metaclass` (no `class` suffix here). And `Metaclass` is an instance of another class, called `Metaclass class` (`class` suffix included), which again must be an instance of something, so it was made an instance of `Metaclass`. (Readers should be forgiven here for thinking they are reading part of the script of Terry Gilliam's *Twelve Monkeys*.) Things get even more intricate if we take into account the inheritance relationships as well (we have been talking only about *instances* until now). There we find that `Object class` is a subclass of `Class`, which is an instance of `Class class`; `Class` is a subclass of `ClassDescription`, and that of `Behavior`, and that of `Object`, and in Squeak we also have `ProtoObject` at the apex of the hierarchy. The situation is presented graphically in Figure 14-2. Note that we do not include any class below `Object` here. The reader can try and combine Figure 14-1 and Figure 14-2 (don't forget to add the metaclasses, starting from `SmallInteger class` and working your way up). It may be that we cannot have too much of a good thing; it is nice to have everything an object, but following the axiom to all its consequences does not necessarily lead to an easily accessible structure. Fortunately, most Smalltalk programmers do not need to be concerned with such matters.

There are other consequences of the "everything is an object" and "everything is carried out by messages" dicta that do concern everyday programmers. Any programmer with a minimum of programming experience would expect that:

```
3 + 5 * 7 == 38
```

would be true, but alas, in Smalltalk the following is true:

```
3 + 5 * 7 == 56
```

The reason is that the binary arithmetic operators are in fact nothing but selectors of the number classes. They are therefore parsed as any other message, and there is nothing dictating that one binary message should have precedence over another. True, it makes sense once you are familiar with the rules of the game. But it still makes one wonder. It is easy to check arithmetic calculations in a workspace in a Smalltalk environment, but it might be better if the programmer never really had to think about such issues.

The Smalltalk environment itself is one of the great innovations of Smallktalk. Back in the 1980s, when graphical displays were a rarity and most programming was done on monochrome text terminals, Smalltalk implemented a graphical user interface (GUI) for a language built on top of a virtual machine. That might have been ahead of its time. Virtual machines had to wait another decade for Java to bring them into mainstream programming. GUIs won the battle, but only when hardware prices came down. In the meantime, an