> The only guarantee of correctness, regardless of whether your language is strongly or weakly typed, is whether it passes all the tests that *define the correctness of your program*. (*http://www.mindview.net/WebLog/log-0025*)

Sometimes it pays to be cautious, though. We may be adding objects to some collection and expecting these objects to obey a certain interface; we would be loath to discover at runtime that some of these objects in fact do not. We can guard against such mishaps by using Smalltalk metaprogramming facilities.

Metaprogramming complements latent typing by allowing us to check that the specified interface is really offered, and giving us the opportunity to react if it is not—for instance, by delegating nonimplemented calls to delegates that we know they implement them, or by simply handling failure gracefully and avoiding crashes. So in the pickaxe book (Thomas et al. 2005, pp. 370–371), we find the following Ruby example, where we try to add song information to a string:

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end
  result << song.title << " (" << song.artist << ")"
  end
```

This is what we would do if we adopted a Java or C# programming style. In Ruby-style duck typing, it would simply be:

```
def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end
```

This code will work with any object that appends using <<; for those objects that do not, we will get an exception. If we really want to be defensive, we can do so by checking the object's capabilities and not its type:

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.respond_to?(:<<)
    fail TypeError.new("'result' needs `<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("'song' needs 'artist' and 'title'")
  end
  result << song.title << " (" << song.artist << ")"
end
```

Smalltalk offers the `respondsTo:` method, defined in `Object`, which we can use to see in runtime whether a given receiver has a given selector.