Pseudo-Classes and Quasi-Classes
Confuse Object-Oriented Programming
Conrad Weisert, March, 1998

Pseudo classes collect non objects
If you attended an introductory presentation on Java
in the past year, you probably heard an enthusiastic
speaker proclaim that Java, in contrast to C++, is a
pure object-oriented language. “In Java everything is
an object.” echo numerous articles and textbooks.
Some Smalltalk proponents make a similar claim.

Of course, that’s nonsense. You can’t write a non-
trivial program in Java without using independent
functions, just as it is in C++. You just have to
package those functions as static class members, and
qualify references to them by the class name.

Indeed some Java classes exist solely as packaging
artifices for collections of non-object-oriented func-
tions. Such classes aren’t true object-oriented classes
at all, since it makes no sense:
• to instantiate an object of that class
• to derive another class from that class

We call such classes pseudo classes to emphasize
that they don’t fit the object paradigm. Examples are
found in Java’s own standard library. No program
ever created a Math or System object.

A pseudo class serves more or less the same role as a
C++ namespace. Whether you consider it a useful
tool in organizing large programs or an awkward
complication, you have to regard it as lying outside
the object paradigm.

Quasi Classes Pollute Bad Programs
Fifteen years ago we saw the phenomenon of quasi-
structured programming (QSP). Naive programmers,
after a short course in “structured programming”
(often presented by an equally naive instructor) would
proudly grind out code that complied with the letter of
what they believed were structured programming’s
rules, while seriously violating structured program-
ing’s spirit and intent. Frustrated managers won-
dered why they never realized the expected break-
through in productivity and quality.

A depressingly similar phenomenon is now spreading
through the object-oriented programming world. The
negative impact of quasi-object-oriented program-
mapping (QOOP), however, is far more severe. QSP
software was never any worse than unstructured soft-
ware. QOOP software, on the other hand, is a lot
worse than a well-designed procedural program.

A question
Last year I published this puzzle in the monthly
Newsletter of the Chicago Chapter of the ACM:

What’s wrong with this class?
class Thing {
  private:
    long value;
  public:
    Thing(const long x=0)
    : value(x) {}  // constructor
    Thing(const Thing& t)
    : value(t.value) {}  // copy constructor
    ~Thing() {}  // destructor
    Thing& operator=(const Thing& rs)
    {value = rs.value;
     return *this;}
    long getValue() const
    {return value;}
    void setValue(const long x))
    {value=x;}
};

Some readers noted various minor problems. Many
were issues of style and taste. Several of the member
function definitions are unnecessary, since the com-
piler supplies them by default. Some programmers
prefer a different sequence of presentation.

Perceptive object-oriented readers noted something far
more serious. The main problem is that our Thing
class serves no purpose at all. The first indication
that something’s fishy is the getValue accessor
function. What does it mean to get the “value” of an
object? In what senses is an object’s value different
from the object itself?

Our suspicions are reinforced when we spot the
setValue method. The combination of these two
nullifies the advantage we get from data hiding. The
class designer took pains to hide the member data
tem value in the private section. That prevented
programs from fetching and storing directly into

copyright 1999, Information Disciplines, Inc.
value or from knowing value’s attributes. So far so good. But what’s the point, if programs can do exactly the same things with the accessor functions?

Although this was a contrived and oversimplified example, I come across just this pattern often, not only in reviewing actual projects but also in textbooks and articles by writers who ought to know better. Some Java gurus, for example, discounting the inability to use ordinary expression syntax on reference objects, see nothing wrong with this:

```java
Money unitPrice, totalDue, shipChg;
int quantityOrdered;

totalDue.setValue(quantityOrdered * unitPrice.getValue() + shipChg.getValue());
```

To those who appreciate the object-oriented paradigm that’s an abomination, and not only because of the clumsy and unreadable syntax.¹ What do we gain in having a Money class at all if any program can do anything it wants to the “hidden” value and if every part of the program knows the units, the base, the range, and the precision of accessor functions which are nothing more than surrogates for the hidden internal representation?

**Larger quasi-classes**

Even worse are composite object classes in which multiple member data items get the same treatment. We see this pattern so often that I’ve coined the term quasi-class for it. A quasi-class is a class that exhibits these properties:

1. For nearly every member data item there’s a corresponding accessor function to get it, such that:
   a. The function takes no parameters.
   b. The returned value is the member data item
   c. The returned value has same type and other attributes as the member data item.

2. For almost every member data item there’s a corresponding function to “set” it, such that:
   a. The function takes a single parameter.
   b. That parameter has the same type and other attributes as the member data item.
   c. The function stores the parameter into the member data item. The return type is `void`.

³ A lack of essential operators and other methods forces user programs to manipulate the object through the get and set accessor functions.

I’ve witnessed C++ and Java programmers routinely churning out such classes according to a sort of mental template. When I ask them to explain their design, they often insist that this is some sort of “canonical form” that all elementary and composite item (i.e. non-container) classes are supposed to take, but they’re at a loss to explain what it accomplishes. They sometimes claim that we need the get and set functions because the member data are private, and, of course, the member data have to be private so that they can be changed without affecting other programs!

In extreme cases, we find written programming standards that actually require that misguided pattern.

A large project can have dozens, even hundreds, of quasi-classes, many of which have dozens of member data items. Those projects are thus burdened with an immense number of components that serve absolutely no purpose. The aggregate number of accessor and related functions in a major project may be close to a thousand!

But maintaining and compiling all those functions isn’t the worst problem. Either the using programs legitimately need to manipulate the member data or they don’t. If they do, then we’re imposing an absurdly awkward syntax (as in the earlier Money example) on them. If they don’t, then we shouldn’t be making it easy for them to do so.

**The bottom line**

Pseudo classes and quasi classes are separate phenomena, both of which undermine the object paradigm. The latter has the greater negative impact. A large application with lots of quasi-classes will be a maintenance nightmare over its life span. Since many such applications are being designed and built by consulting or contract development firms, that maintenance burden will likely fall upon those who had little to do with the design decisions. Many of those maintenance programmers will have even less appreciation of the object-oriented paradigm than the original designers did.

Management will likely conclude, as they already have in one “object-oriented” project after another, that OOP offers few if any practical benefits. Can we blame them?

¹ For a more disciplined way of accomplishing the same thing in Java, see Conventions for Arithmetic Operations in Java on www.idinews.com.