LETTER TO THE EDITOR[1]

In their recent article [1], Al-Ahmad and Steegmans elaborate on a variant of inheritance they call *specialization inheritance* and its implementation through new OOPL features. To motivate their approach, they resort to a popular example involving rectangles and squares, which in the past has led to lengthy discussions (cf, eg, [2]).

In their recent book, D'Souza and Wills, regular columnists to this Journal, state:

> *It is now widely accepted good practice that nearly every class should either be an abstract class (prohibited from having instances but possibly with a partial implementation) or a final class (prohibited from having extensions).* [3]

I do not know why they say this, but let us, for the moment at least, subscribe to this point of view.

Following the quoted principle, the class hierarchy of Figure 1 should be transformed to that of Figure 2. Clearly, this leads to significantly more classes and therefore seems to counteract the idea of code reuse, the primary objective of Al-Ahmad and Steegmans. However, as it will turn out, it paves the way for an implementation that is elegant, conceptually sound, and readily realized. Let us see.

```
ClosedFigure                    ClosedFigure
  Quadrangle                      Quadrangle
    Parallelogram                   Parallelogram
      Rectangle                       Rectangle
        Square                          Square
  Ellipse                               NonSquareRectangle
    Circle                            NonRectangularParallelogram
                                    OtherQuadrangle
                                  Ellipse
                                    Circle
                                    NonCircularEllipse
                                  OtherClosedFigure
```

Figure 1: Class hierarchy used as an example in [1]; subclasses indented

Figure 2: Same as Figure 1 with necessary final (leaf) classes added; abstract classes in italics

Because the intrinsic regularity of the more specialized subclasses of closed figures lets them make do with fewer instance variables than their more general ancestors, it is a good idea to start designing the classes bottom up. Therefore, we assign one instance variable, a, which is to denote the length of one side, to class *Square*, and two instance variables, a and b, to class *NonSquareRectangle* which are to hold the different side lengths of the instances of all other rectangles.

Quite obviously, *Square* and *NonSquareRectangle* share the instance variable a, which can be factored out to *Rectangle*. Note that leaving b in *NonSquareRectangle* saves us from having to drop it from the definition of *Square*, which is a major concern in the work of Al-Ahmad and Steegmans. However, it also leaves us with a class hierarchy with which no access to side b is possible for non-square rectangles assigned to a variable of type *Rectangle*.

The remedy is simple. We just add virtual access methods to side b in class *Rectangle*, and implement them as access to b in class *NonSquareRectangle* and as access to a in class *Square*. Doing so is conceptually sound as squares are indeed rectangles that have four sides all of

---

which are of equal length (access to sides *c* and *d* would, presumably, be required by inheritance from class *Quadrangle*). At the same time it allows us to define and implement class-typical operations, including *Surface*, in class *Rectangle* without any need for overriding in its subclasses. This may be considered elegant.

Accessing instance variables through access functions is generally considered good practice. It is enforced in Smalltalk and suggested by special language constructs in Delphi (in the form of *properties*) and Java (through *interfaces*, which do not declare instance variables). Virtual access methods are also the key to the solution offered by Al-Ahmad and Steegmans; however, their adhering to the class hierarchy of Figure 1 forces them to introduce new language constructs that will not be easy to establish. By contrast, the approach presented here should be readily implementable in any OOPL.

While some conceptual aspects worthy of consideration could be added [2], a few other points made by Al-Ahmad and Steegmans need clarification.

- It is enforced, either statically through compile-time errors or dynamically through run-time errors, by every OOPL I know about (and by any theory of OOP, for that matter) that all instances compatible with a certain type obey the protocol specified by that type, ie, respond to the messages listed in the declaration of the type. While there are compelling theoretical reasons for this [4], it also has an important practical implication: it ensures safe extensibility of programs. In C++, types are implicitly defined by classes, and the instances assignment compatible with a type are the instances of the defining class and all its subclasses (principle of substitutability). Java goes one step further and adds separate type definitions, called interfaces, that classes may choose to implement independent of the class hierarchy. The suppression of operations in subtypes, however, is a highly questionable desideratum.

- It is generally agreed that the *extension* of *every* subtype (ie, the set of instances that are assignment compatible with that type; note the overloading of the term *extension*) is a subset of the extensions of all its supertypes [4, 5]. This guarantees the substitutability of instances of subtypes for instances of its supertypes; it is *the* principle behind inheritance. That the *intension* (ie, the specification or declaration) of one subtype is an extension (in the sense that it adds more code) of that of its supertype, while the intension of another subtype is a restriction (in the sense that is restricts the types of members), is an entirely different pair of shoes. It is even possible that the declaration of a subtype is both and extension and a restriction of the declaration of its supertype [4]. Therefore, in the context of subtyping the distinction between extension inheritance and specialization inheritance is conceptually meaningless (even though it may have some implementational import).

- While the type of 3D points is indeed an extension of the type of 2D points in the spirit of [6], it should not be considered a subtype, simply because semantically the set of 3D points is not a subset of the set of 2D points. Any OOD making this kind of assumption is likely to run into serious trouble (because substitutability is not enforced by the semantics of the types).

Personally, I consider the design principle quoted at the beginning of this letter, especially if combined with single inheritance, extremely helpful, basically because

- it makes class hierarchies represent conceptually clean, strict partitions of the problem domain (taxonomies); and

- it provides the skeleton for practically appealing implementations, even under the perspective of code reuse.

-

References

[1] W Al-Ahmad, E Steegmans "Improving support for specialization inheritance" *Journal of Object-Oriented Programming* 11:8 (1999) 29–36.

[2] JA Grosberg "Comment on objects considered harmful" *Communications of the ACM* 36:1 (1993) 113–114.

[3] DF D'Souza, AC Wills *Objects, Components and Frameworks with UML* (Addison-Wesley 1998) 146.

[4] P Wegner "The object-oriented classification paradigm" in: P Shriver, P Wegner (eds) *Research Directions in Object-Oriented Programming* (MIT Press 1987) 479–560.

[5] DM Papurt "Generalization and polymorphism" *Report on Object Analysis & Design* 2:5 (1996) 13–16.

[6] N Wirth "Type extensions" *ACM Transactions of Programming Languages and Systems* 10:2 (1988) 204–214.

Address for correspondence:

Dr. Friedrich Steimann
Institut für Rechnergestützte Wissensverarbeitung
Universität Hannover
Lange Laube 3
30159 Hannover
Germany

steimann@acm.org