Friedrich Steimann

Abstract Class Hierarchies, Factories, and Stable Designs

Much of the debate about the general aptness of class hierarchies is rooted in the different objectives taxonomists and implementers are thought to pursue. Designers of conceptual hierarchies tend to embrace Aristotle's principle of *genus et differentiae* leading to a taxonomic hierarchy of categories or types [7], while those with implementation in mind focus on the reuse of class definitions and polymorphism as made possible by subclassing and inheritance. This has led to an extensive discussion (see [1, 4, 5, 8]) as to whether Square should be a subclass of Rectangle or vice versa, a dilemma that is, of course, precedential in character.

Despite the different perspectives there appears to be a broad consensus that, in principle at least,

- both a conceptual type and a class (as a programming construct) are intensions the extensions of which are sets of instances; and
- the extensions of subtypes are subsets of the extensions of

their supertypes, so that the instances of a subtype can occur wherever instances of its supertypes are expected (principle of substitutability).

Depending on the programming language, the latter may not be the case for class hierarchies so that conceptual type hierarchies and class hierarchies are not generally isomorphic to each other [6]. However, as Grosberg rightfully observed [4], the discrepancies can easily be resolved by adhering to one simple rule: by requiring that only the leaf classes can have instances.

Abstract Class Hierarchies

This rule is not as arbitrary as it may seem. In fact, it only paraphrases a common constraint on subtyping, namely that the extension of a supertype is totally covered by the extensions of its subtypes, thus rendering the supertype a mere abstraction. For example, applied to the class hierarchy of Figure 1a, it is implied that all instances of type Person must either be an instance of class Female or of class Male.

Following this principle, the Rectangle/Square dilemma is resolved as shown in Figure 1b), where OtherRectangle denotes the set of rectangles that are not squares. Surely, this is going to affront many system modelers and most implementers: why waste the name Rectangle for an abstract class which cannot have instances, and why introduce an additional class oddly named OtherRectangle which will create most of the instances of Rectangle? First, not having class OtherRectangle is a bit like having classes Person and Female, but not Male. Second, OtherRectangle could, of course, just as well be named NonSquareRectangle—the point here is there is always a sibling class that holds the remainder otherwise assigned to the superclass. And third, when creating a particular rectangle, its clients need not see or know about (unless they desire to) the distinction between Square and OtherRectangle—they simply resort to a factory.

Figure 1. (subclasses are indented, leaf classes underlined).
a) Female and Male are the only subclasses of Person
b) rectangles are either Squares or OtherRectangles

```
a)Person   b)Shape
    Female      Rectangle
    Male          Square
                  OtherRectangle
                ...
```

Factories

A factory is an object-oriented programming construct providing for the creation of instances without specifying their concrete classes. Factories come in many different guises, the most common of which have been stereotyped in the form of design patterns [2, 3]. Here we think of a factory as an abstract class whose creator methods (called factory methods) return instances of its concrete subclasses. In the geometrical shape example, squares and (nonsquare) rectangles might be created by calls to factory methods of class Shape as shown in Figure 2.

The clients of the hierarchy, cognizant only of class Shape, will not know or need not care about the actual type of the instance they get, but nevertheless (through dynamic binding) receive all the benefits of the different, possibly optimized implementations of methods for classes Square and OtherRectangle, such as the calculation of the area.

One may object: what if I stretch a square in one dimension? Does that not imply instance migration? Well, what if I shear a rectangle? Indeed, stretching and shearing should be viewed and implemented as what they actually are: mathematical operations that return new instances. In this light, every operator is a small factory method returning a new instance of a class determined solely by the operands (including the implementor) and the operator itself. The same principle naturally applies to hierarchies of numbers, collections, and so forth, with plenty of opportunities to exploit the efficiency and

Figure 2. Calls to a factory class creating instances of the appropriate type.

```
Shape.rectangle(120,60) // (width, height)
   // creates a new instance of class OtherRectangle
Shape.square(80)
   // creates a new instance of class Square
Shape.rectangle(80,80)
// also creates a new instance of class Square
```

maintainability gains offered by a clean partitioning of the problem domain. For instance, depending on its operands, the division of two integers may return an integer or a (noninteger) fraction.

Stable Designs

While the practical benefits of conceptually sound class hierarchies are still arguable, there is another, very pragmatic reason to enforce the rule of letting only leaf classes have instances: it protects the rest of the class hierarchy from ad hoc alterations made to individual class definitions. Given that most of the many changes that become necessary in the course of system evolution pertain to the behavior of instances of individual classes, the propagation of these changes (through inheritance) to other classes is not generally desired. However, especially if class hierarchies are big and used by many clients, the existence of and consequences for descendant classes are not immediately realized, making inheritance a mixed blessing. By designing the class hierarchy as a hierarchy of abstract classes and by letting its clients manipulate only the concrete classes attached as leaves, the effect of modifications needing to be made by the clients is always confined to the instances of single classes. The need for a (partial) redesign of the class hierarchy because of practical requirements is thus greatly reduced. C

Freidrich Steimann (steimann@acm.org) is a research assistant at the Universität Hannover, Germany.

References

1. Baclawski, K. and Indurkhya, B. The notion of inheritance in object-oriented programming. *Commun ACM 37*, 9 (Sept. 1994), 118–119.
2. Cooper, J.W. Using design patterns. *Commun ACM 41*, 6 (Jun. 1998), 65–68.
3. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.
4. Grosberg, J.A. Comment on considering 'class' harmful. *Commun. ACM 36*, 1 (Jan. 1993), 113–114.
5. Halbert, DC. O'Brien, P.D. Using types and inheritance in object-oriented programming. *IEEE Softw. 4*, 5 (May 1987), 71–79.
6. LaLonde, WR and Pugh, JR. Subclassing π subtyping π is-a. J. *Object-Oriented Program.* (Jan. 1991), 57–62.
7. Sowa, J.F. *Conceptual Structures: Information Processing in Mind an Machine*. Addison-Wesley, Reading, Mass., 1984.
8. Winkler, J.F.H. Objectivism: 'class' considered harmful. *Commun. ACM 35, 8* (Aug. 1992), 128–130.