Editor: Bertrand Meyer, Interactive Software Engineering, 270 Storke Road, Suite 7, Goleta, CA 93117; phone (805) 685-1006; fax (805) 685-6869; Internet ot-column@eiffel.com

The many faces of inheritance: A taxonomy of taxonomy

Bertrand Meyer
Interactive Software Engineering

One of the most important considerations in constructing object-oriented software is the methodology of inheritance: how to use this mechanism well. In preparing the second edition of my book, *Object-Oriented Software Construction* (Prentice Hall, 1988), I wrote a detailed discussion of the methodology of inheritance. One of the prerequisites to the work on the corresponding chapter (which can be consulted in draft form at http://www.eiffel.com, following the link to "Object Technology Column") was to classify the various forms of inheritance — to construct a taxonomy of taxonomies. This column presents the taxonomy, comprising 12 kinds of inheritance usage grouped into three broad categories.

A broad-minded view

A widely circulated view holds that inheritance should only be used for strictly limited purposes, and that any form of "implementation inheritance" or "inheritance for code reuse" is bad. I find this view too restrictive. It also lacks the strong theory that should support any such indictment (as shown by the first and most famous of all, Edgar Dijkstra's 1968 excommunication of the `goto`, based not on opinion but on a detailed theoretical argument and a comprehensive model of software development).

My own view of inheritance is broad. My colleagues and I have used all 12 of the categories extensively. I find them both theoretically legitimate and practically indispensable.

In this column, however, we have to set aside such differences. If your attitude toward inheritance is more restrictive, perhaps this classification will help you better appreciate the categories you accept and more lucidly criticize those you reject.

Proper and improper uses

A broad-minded view of inheritance does not mean that anything goes! It is easy and sometimes tempting to misuse the mechanism. In general, inheritance is applicable only if you can seriously argue for the presence of an "is" relation between the instances of the heir and parent classes. This guideline is broad enough to include some of the more controversial uses, but limiting enough to exclude obvious mistakes.

Improper uses of inheritance tend to fall into three categories:

- *"Has" relation with no "is" relation.* Over the years I have heard or seen a few similar ones, often as purported examples of multiple inheritance, such as *APPLE_PIE* inheriting from *APPLE* and from *PIE*. Another example, reported by Adele Goldberg, is *ROSE_TREE* inheriting from *ROSE* and from *TREE*.
- *Taxomania* (an abbreviation for "taxonomy mania"). This is a typical result of beginner's enthusiasm. Beginners often add useless intermediate nodes in the inheritance structure as a substitute for simple Boolean properties (such as gender for people records) or properties that have a few fixed values (such as the color of a traffic light). The theory of abstract data types supplies the appropriate guidance here: Introduce a new class only if it provides significant new or modified functionality in the form of query or command operations — features.
- *Convenience inheritance.* The developer sees some useful features in a class and inherits from that class simply to reuse these features, without the proper "is" relationship between the corresponding abstractions — or in some cases without adequate abstractions at all.

General taxomony

Figure 1 shows the taxonomy's general structure. The classification is based on the observation that every software system reflects a certain external model, which is in turn connected with some outside reality in the application domain. So it distinguishes between:

- *Model inheritance*, which reflects "is-a" relations between abstractions in the model.
- *Software inheritance*, which expresses relations within the software itself rather than in the model.
- *Variation inheritance*, which describes a class by how it differs from another class (a special case that may pertain either to the software or to the model).

The definitions assume that the parent class is called *A* and the heir class *B*. Each definition will state which of *A* and *B* is permitted to be *effective* and which *deferred*. (A class is effective if it is fully implemented; deferred if one or more of its features are specified but not implemented.) The examples that follow use Eiffel conventions.

Subtype inheritance

A and *B* represent certain sets *A′* and *B′* which are external objects. *A* is deferred, *B′* is a subset of *A′*, and the set modeled by any other subtype heir of *A* is disjoint from *B′*.

This definition assumes that the software relies on a model of some external system, for example, some aspect of a company's business or some part of the physical world.

Subtype inheritance is the inheritance form that is closest to the hierarchical taxonomies of botany, zoology and other natural sciences. A typical software example is *CLOSED_FIGURE* ← *ELLIPSE*. We insist that the parent, *A*, must be deferred, so that it describes an incompletely specified set of objects. *B*, the heir, may be effective or deferred.

View inheritance

B describes the same abstraction as *A*, but viewed from a different angle.

View inheritance is an advanced technique, open to misuse, so I will stay away from it in this presentation. Please see the detailed discussion in the Web document.

Restriction inheritance

The instances of *B* are those instances of *A* that satisfy a certain constraint, expressed if possible as part of the invariant of *B* and not included in the invariant of *A*. Any feature introduced by *B* should be a logical consequence of the added constraint. *A* and *B* should be both deferred or both effective.

Many mathematical examples indeed fall into this category, including *ELLIPSE* ← *CIRCLE*, where the extra constraint is that the two focuses of an ellipse are merged into one point for a circle. The second sentence of the definition is meant to avoid mixing this form of inheritance with others and to limit new features, if any, to those that directly follow from the added constraint. For example, class *CIRCLE* has a new feature *radius* which satisfies this property: In a circle, all points have the same distance from the merged center. This distance deserves the status of a feature of class *CIRCLE*, whereas the corresponding notion in class *ELLIPSE* (the average of the distances to the two focuses) is probably not significant enough to yield a feature. Because the only conceptual change from *A* to *B* is to add some constraints, the classes should be both deferred or both effective.

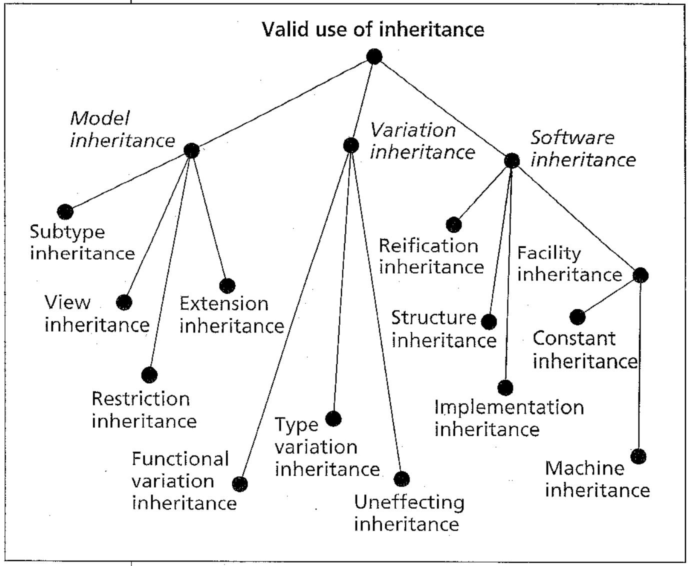

Figure 1. Inheritance taxonomy.

Extension inheritance

B introduces features not present in *A* and not applicable to direct instances of *A*. *A* must be effective.

The presence of both the restriction and extension variants is one of the paradoxes of inheritance. Extension applies to features, and restriction (and more generally specialization) applies to instances, but this does not eliminate the paradox. The problem is that the added features usually include attributes. So if we take the naïve interpretation of a type (as given by a class) as the set of its instances, then it seems the subset relation is the wrong way around! Assume for example

```
class A feature a1: INTEGER end
class   B inherit
        A
feature
        b1: REAL
end
```

If each instance of *A* represents a singleton and each instance of *B* is a pair containing an integer and a real number, then the set of pairs *MB* is not a subset of the set of singletons *MA*. In fact, the subset relation is in the reverse direction: There is a one-to-one mapping between *MA* and the set of all pairs having a given second element, for example *0.0*.

This discovery that the subset relation seems to be the wrong way may make extension inheritance look suspicious. An example (again from Adele Goldberg) is an early object-oriented library that had *RECTANGLE* inheriting from *SQUARE* rather than the other way around: *SQUARE* has a side attribute; *RECTANGLE* inherits from *SQUARE* and adds a new feature, other_side. This was criticized and soon corrected.

But we cannot dismiss the general category of extension inheritance. In fact its equivalent in mathematics, in which a notion is specialized with the addition of completely new attributes (or their mathematical counterparts) is frequently used, and everyone considers it perfectly legitimate. A typical example is the notion of "ring", specializing the notion of "group". A group has a certain operation, say +, with certain properties. A ring is a group, so it also has + with these properties, but it adds a new operation, say *, with extra properties of its own. This is not fundamentally different from introducing a new attribute in an heir software class.

In object-oriented software, in fact, this happens often. In most applications, of course, *SQUARE* should inherit *RECTANGLE*, not the reverse; but it is not difficult to think of more legitimate examples. For example a class *MOVING_POINT* (for cinematic applications) might inherit from a purely graphical class *POINT* and add a feature, *speed*, to describe the speed's magnitude and direction.

Functional variation inheritance

B redefines some features of *A*, and some of the redefinitions affect feature bodies, not just signatures. *A* and *B* are either both deferred or both effective, and *B* must not introduce any new features except for the direct needs of the redefined features.

Functional variation inheritance is used when we want to adapt an existing class without affecting the original class and its clients. This may at first look like a form of hacking: twisting an existing class to fit it to a slightly different purpose. At least it avoids the dangers of directly modifying the existing software. But if you do have access to the source code of the original class, you should examine whether it is not preferable to reorganize the inheritance hierarchy by introducing a more abstract class of which both *A* (the existing variant) and *B* (the new one) will both be proper descendants with peer status.

Type variation inheritance

B redefines some features of *A*, and the redefinitions affect only signatures. *A* and *B* are either both deferred or both effective, and *B* must not introduce any new features except for the direct needs of the redefined features.

Type variation inheritance is necessary only if some of the original signatures did not make enough use of anchored (**`like ...`**) declarations. For example, suppose the *SEGMENT* class of an interactive drawing package has the function

```
perpendicular: SEGMENT is
     -- Segment of same length
     -- and same middle point,
     -- rotated 90 degrees
     ...
```

and you want to define an heir, *DOTTED_SEGMENT*, to draw a dotted line. To accomplish this, perpendicular should return a result of type *DOTTED_SEGMENT*, so you must redefine the type.

This could all be avoided if the original returned a result of type **`like`** *`Current`* and you had access to the source of the original. You could then update the type declaration without adversely affecting existing clients. If for some reason this is not possible, the ability to redefine the type this way can save your neck.

Uneffecting inheritance

B redefines some of the effective features of *A* into deferred features.

This form of inheritance is not — and should not be — common. It goes against the normal direction of inheritance — we usually expect *B* to be more concrete and *A* more abstract. Beginners should stay away from uneffecting, but it may be justified in two cases:

- In multiple inheritance, you want to merge features inherited from two different parents and both are effective. You uneffect one of them so that the other's implementation will take precedence.
- You find a reusable class that serves your needs, except that it is too concrete. You can use uneffecting to remove the unwanted implementations.

Reification inheritance

A represents a general kind of data structure and *B* represents a partial or complete implementation choice for that data structure. *A* is deferred; *B* may still be deferred — leaving room for further reification through its own heirs — or it may be effective.

For example, the deferred class *TABLE* describes tables of a very general nature. Reification leads to heirs *SEQUENTIAL_TABLE* and *HASH_TABLE*, still deferred. Final reification of *SEQUENTIAL_TABLE* leads to effective classes *ARRAYED_TABLE, LINKED_TABLE*, and *FILE_TABLE*.

Structure inheritance

A, a deferred class, represents a general structural property, and *B*, which may be deferred or effective, represents a certain type of object possessing that property.

Usually the structural property represented by *A* is a mathematical property. For example, the class *COMPARABLE* in the standard Eiffel library is equipped with operations like ***infix*** "<" and ***infix*** ">=," representing objects to which a total order relation is applicable. A class that needs an order relation of its own, such as *STRING*, will inherit it from *COMPARABLE*.

It is common for a class to inherit from several parents in this way. For example, class *INTEGER* inherits from *COMPARABLE* and from class *NUMERIC* (with features such as infix "+" and infix "*").

Implementation inheritance

B obtains from *A* a set of features (other than constant attributes and once functions) necessary to implement the abstraction associated with *B*. Both *A* and *B* must be effective.

A common case is the "marriage of convenience," based on multiple inheritance, in which one parent provides the specification (reification inheritance) and the other the implementation (implementation inheritance). For example, the EiffelBase library class *ARRAYED_STACK* inherits its specification from *STACK* and its implementation from *ARRAY*. This form of inheritance is sometimes criticized, but has turned out to be extremely useful in practice.

Facility inheritance

A exists solely to provide a set of logically related features for the benefit of heirs such as *B*.

Here the parent, which is a collection of useful features meant only for use by its descendants, becomes an heir of another chiefly for the benefit of using these features: There are two common variants:

- Constant inheritance, in which the features of *A* are all constants or once functions describing shared objects. Both *A* and *B* are effective.
- Machine inheritance, in which the features of *A* are routines, which may be viewed as operations on an abstract machine. *B* should be at least as effective as *A*.

For example, class *EXCEPTIONS*, a utility class that provides facilities for detailed access to the exception-handling mechanism, is meant to be inherited by classes that need those facilities. Sometimes, facility inheritance uses only one of the two variants, but in other cases like *EXCEPTIONS*, the parent class provides both constants (such as the exception code *Incorrect_inspect_value*) and routines (such as trigger to raise an exception).

One mechanism, or more?

The variety of uses of inheritance may lead to the impression that we should have several language mechanisms to cover the underlying notions. Several authors have suggested separating between module inheritance, which is essentially a tool to reuse existing features, and type inheritance, which is essentially a type-classification mechanism.

Such a division seems to cause more harm than good, for several reasons.

First, recognizing only two categories is not representative of the variety of uses of inheritance. No one would advocate introducing 12 different language mechanisms.

Second, the practical effect of a division would be to raise useless methodological discussions. Assume you want to inherit from an iterator class. Should you use module inheritance or type inheritance? One can find arguments to support either answer. You will waste your time trying to decide between two competing language mechanisms. The contribution of such reflections to the only goals that count—quality software and fast delivery—is exactly zero.

Such a division would also bloat the language. And whenever we introduce new mechanisms into a language, they interact with the rest, and with each other. Do we prohibit a class from both module-inheriting and type-inheriting the same class? If so, we may be just vexing developers who have a good reason to use the same class in two ways; if not, we open up a whole can of new language issues—name conflicts, redefinition conflicts, and so on.

There is only one serious objection to the use of a single mechanism: The extra complication it imposes on the task of static type checking. Solutions to this issue place an extra burden on compilers, which is always justifiable if the burden is reasonable and the effect is to facilitate the developer's task.

The ability to use only one inheritance mechanism for both module and type inheritance is not the result of a confusion of genres. It is the direct consequence of the very first decision of object-oriented software construction: The unification of module and type concepts into a single notion, the class. If we accept classes as both modules and types, then we should accept inheritance as both module accumulation and subtyping.