

Bertrand Meyer

Overloading vs. Object Technology

C++ programmer said to Rabbi Shammai: "I promise to try object technology if you can explain the secret of objects while standing on one foot." Rabbi Shammai whacked him on the head with a 10-by-2 ruler. He went to Rabbi Hillel who, standing on one foot, answered: "No overloading. That is the secret of objects. All the rest is commentary."

Some commentary now. The beauty of object technology (OT) is that for all the variations, refinements, codicils, and consequences—including fifteen years of the *Journal of Object-Oriented Programming* articles—everything in the end rests on a single idea: class. A class is an association between some **names** and some **operations**, called "features" in Eiffel (and known as "members" in several other languages). In a class *POINT* describing points in a plane, the names may include *x*, *y*, *ro*, *theta*, *move*, and *rotate*; the features may include operations to return a point's cartesian and polar coordinates, move it by a certain displacement, and rotate it around the origin by a certain angle. The association between names and features is one-to-one: *x* denotes the operation that returns the horizontal coordinate; *ro* the one that returns the distance to the center; *rotate* the rotation operation.

Enforcing a one-to-one correspondence keeps everything simple and manageable. It makes the class readable and avoids confusion: if you see a feature name *f* and do not know what it means, you only have to search until the first feature declaration with that name, and stop there; you know that you have found what you are looking for, and do not need to worry about some competing definition. Easy, simple, and comforting.

THE PRINCIPLE

No-Overloading Principle

Different things should have different names.

The rule is simple:

Human languages do not quite follow that rule. This can be a source of riddles, as in "time flies like an arrow." Amusement apart, it is also a source of confusion, not appropriate for a programming or specification language that is designed for precise statement of intent. Within a single syntactic scope such as a class, just choose different names.

Why indeed give the same name to two different things? Names are not an endangered species, and there is no tax on keystrokes. Even in Eiffel, where we like giving features clear, meaningful, pronounceable names—I have never been able to understand why others use *btnClick* or even *btnClk* when it is so simple to write *button_click*, and say it loud when needed—we have endless possibilities from words and their combinations. If it is two different things, call them by different names. Appending a character to a string, or concatenating another string at the end, are not the same operation by any stretch of imagination. Why pretend that they are, and in the process confuse the poor program reader? Just use

extend (c: CHARACTER) -- Add c at end. ensure count = old count + 1 item (count) = c

for adding a character, and

```
append (s: STRING)

-- Concatenate characters of s at end.

require

s /= Void

ensure

count = old count + s.count

equal (substring (old count + 1, count), s)
```

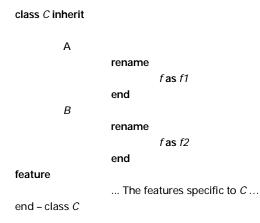
for concatenating a string. The different contracts confirm that these are different operations. Features to concatenate the string representation of an integer, a real number, etc., will similarly have different names to reflect their differences.

Note that this practice goes well with the rule, part of the Eiffel style guidelines, of not over-qualifying feature names. A beginner will sometimes be tempted to include the target type's name into a feature name, calling the feature *extend_to_string*. The style rules disallow this practice since the extra qualification leads to unnecessarily long names: in any actual use of the feature, such as *my_string.extend ('X')*, the type of the target, here *my_string*, unambiguously specifies the class to which the feature belongs (or an ancestor of that class).

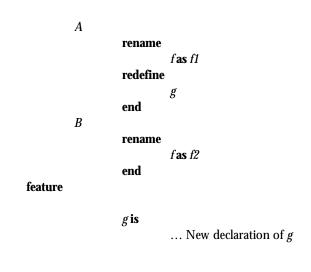
Bertrand Meyer is CTO of Interactive Software Engineering, which just released ISE Eiffel 5.0, the most important new version since ISE Eiffel 3 in 1993. (http://www.eiffel.com). His latest book is *The .NET Training Course* (Prentice Hall). He may be reached at Bertrand_Meyer@eiffel.com.

RENAMING

The No-Overloading Principle (meaning that a class establishes a one-to-one correspondence between feature names and features) has a strong consequence on multiple inheritance: if a class C has two parents A and B, and both have a feature called f, you can't leave things as they are; this would be as bad as introducing two features called *f* in *C* itself. The Eiffel technique relies on the observation that there is nothing wrong with A and *B* taken individually: each of them, to be valid, must have provided its one-to-one correspondence. There is also no problem (in the absence of further complications, such as might result from repeated inheritance) if C wants to inherit both classes and hence inherit both the features called f. These are different features; the only issue is an unfortunate naming conflict. To resolve that issue, it suffices to rename one or both inherited features, at the point of inheritance in C. This is the well-known renaming technique of Eiffel:



class C inherit

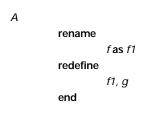


end – class C

... Features specific to $C \dots$

Redefinition changes the feature, not its name. Renaming changes the name, not the feature. In some cases you'll want both:

class C inherit



Now, as a result, *C* defines its own name-feature correspondence, one to one. The power of renaming extends beyond this case: it enables you to adapt the names of inherited features to the context of the new class. Even if a parent's feature is useful to the clients of *C*, its name may not be the most appropriate for those clients, who see the feature in a different light, as part of the abstraction represented by *C*, not its parent. Each class is a machine, whose designer should have full power to define both the functionality (the features) that he deems most useful to the clients, and the exact form under which the clients will get that functionality, including names most appropriate to their needs. Renaming helps the designer achieve that goal.

Past the **rename** clauses, the inherited features are known in the new classes under their new names. This leads to the notion of the **final name** of a feature: its declared name if it is immediate (introduced in the class itself); its original name if it is inherited and not renamed; the new name if it is inherited and renamed. Every reference to a feature of a class, from anywhere in the software text — the feature's class, or another—uses its final name.

The designer will also, in some cases, want to change the inherited features themselves. This is the purpose of redefinition: Note the use of the final name for the redefinition. This view of each class as a professional-quality machine under the precise control of the machine's designer is central to the Eiffel method.

WRONG CRITERION

What is puzzling in languages that support overloading is the criterion they use to distinguish competing features with the same name—through their type signatures. The rule is that it is legal to give two features the same name if and only if at least one of the argument types differs. But that criterion is irrelevant—two competing features may well have the same argument types. For example our *POINT* class might have features *reset_cartesian* and *reset_polar* that reset a point's coordinates to given values. This is a typical situation where overloading, for people attracted to this idea, would seem appropriate. Unfortunately for them, even in such a rudimentary example it doesn't work since the argument types happen to be the same in both cases: both *reset_cartesian* and *reset_polar* will take two arguments of *REAL* type. The signature criterion has nothing to do here.

In such a case you could in principle name the features differ-

ently, but C++ and its successors such as Java and C# don't leave you that possibility. In its overloading zeal, C++ forces you to give all "constructors" of a class—the procedures used to initialize its instances—the same name, which must be the name of the class. Calling everything the same certainly saves efforts of imagination, but the results are strange. In our example, *POINT* would not only be the class name but also the name of all the constructor routines. How then do you offer alternative ways of creating a point, one by providing the cartesian coordinates and one by providing the polar coordinates? You do not. It is just not possible. Here, overloading appears not only as useless and potentially confusing, but as a facility that severely limits the power of expression of the language and prevents the use of commonly useful patterns.

In Eiffel, of course, constructors (creation procedures) are named; you create an object through

create my_point.make_cartesian (1, 0)

where *make_cartesian* is one of the procedures of the class, and is listed in the Creation clause at the beginning of the class:

```
class POINT create
make_cartesian, make_polar, default_create
feature
... Declaration of features, including
make_cartesian and make_polar...
```

Note that creation procedures such as *make_cartesian* and *make_po-lar* are otherwise normal procedures; by setting their export status, you can elect to have them usable for creation only, for resetting of an existing object (playing the role of *reset_cartesian* and *reset_po-lar* as posited above), or both. Procedure *default_create* is available in all classes; you can redefine it in any class to override the default initializations performed in procedure-less creations of the form

```
create my_point
```

formally understood as an abbreviation for **create** *my_point.de-fault_create*. If the class has no Creation clause, it is understood to have one listing only *default_create*, so that the basic form **create** *my_point* is permitted by default.

I have briefly described this Eiffel mechanism, using named creation procedures, because it seems the obvious and simple thing to do. The reliance on overloaded non-named constructors is hard to justify.

INTRODUCING INHERITANCE

With inheritance, the use of overloading appears even more confusing. Consider a simple pair of inheritance links (see Figure 1).

We will use names reminiscent of the corresponding types. In class *A*, feature *f* is overloaded with two variants accepting arguments of types *X* and *Y*. Let's also assume that *B* redefines (overrides) these versions. The following polymorphic assignments are possible for a1 of type *A*, x1 of type *X* and so on:

```
a1 := b1
x1 := y1
```

Now consider the corresponding calls:

a1.f (x1)	
a1.f (y1)	
b1.f (x1)	
b1.f (y1)	

In an OO environment, they should all give the same result, since *a1* and *b1* are both attached to the same object, and *x1* and *y1* both attached to another single object; but overloading complicates everything. (If you program in a language with overloading, can you tell—Quick! No textbooks! No language lawyers!—which variants are permitted? What does each do?)

Overloading simply does not go with inheritance and its associated techniques of redefinition, polymorphism, and dynamic binding. If you are happy to sacrifice inheritance, then you may consider overloading (as in Ada 83), although enough unpleasant consequences remain that I would not advise following that route. But with OO techniques it becomes impossible.

GOVERNMENT BY WARNING

To see the consequences of the systematic reliance on overloading and how it clashes with inheritance, consider the following mechanism, known as "version management," of C#.

If a feature of a class has a name that conflicts with the name of an inherited feature, it must normally be declared as either "**override**" or "**new**." The first case is the normal OO mechanism of redefinition. The second is only possible because of overloading, and requires that the argument signature be different from the signature of the inherited version.

So far so good, or rather (in light of the previous discussion) so far so bad. But now comes a special convention, designed to address the case of delivering to your customers a new version of a class *A* with a new feature called *f*. That name was not used in previous versions of *A*, so perhaps one of your customers had written a descendant class *B* where he introduced a feature of his own, called *f*. But now this name conflicts with that of an inherited fea-

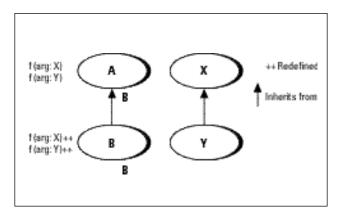


Figure 1. Overloading clashes with inheritance.

ture. Since there is no direct equivalent of renaming, you should normally go back to *B* and change it to declare *f* as **new**. But then it means that *B* does not work "as is" any more. To facilitate the upgrading process, the language lets you omit any **new** or **override** specification, with the understanding that an absent specification will have the same effect as a **new** declaration, and the requirement that the compiler must produce a warning.

This is disturbing. First, it is (at least as far as I know) a first in the history of programming languages: the first case of defining a language's semantics through compiler warnings. Semantics should be defined very precisely, since it conditions the validity of programs and the effect of valid programs. The 70 or so "validity rules" of Eiffel¹ are all of the form,

"A construct of the form, ..., is valid if and only if, ..."

with a list of painstakingly precise conditions telling you exactly when the compiler will reject your attempts ("only if...") and when it *must* accept them ("if"). Relying on an operational description of what "the compiler" will do would be bad enough; relying on a warning is worse.

Then, in practice, where do warnings go? They may very well be ignored. We have all seen cases of compilers producing hundreds of warnings, some meaningful and some not; after a while programmers just brush them aside. When partisans of languages that permit potentially dangerous conversions (casts), such as C and C++, argue with users of strongly typed languages, they point out that really bad type errors will be caught by compiler warnings or *lint*-like tools; but it is easy to show by example that such tools produce such an ocean of unjustified warnings that the few important ones lost in their midst are easy to miss.

This convention, "saying nothing means new plus a warning," seems to be an effort at addressing a minor potential problem by causing major potential damage. How real is the minor problem anyway? If one of your classes inherits from A and you get a new version of A, it would be very foolish to accept the new version blindly; you should check that it does not affect the assumptions that your class made about its parents. In Eiffel, the subcontracting rules help you in this process, by guaranteeing that the descendant's semantics is compatible with the original's: the class invariant is inherited, and in the case of overriding a feature, its properties must remain compatible (precondition kept or weakened, postcondition kept or strengthened). In other languages you don't have this, so it's even more necessary to check that the decision to inherit from A—a strong commitment!—is still legitimate with the new version. Occasionally having to adapt a newly clashing feature name is a small part of the process. In fact, there is further good reason to require manual checking of such cases: ignoring the case of programmers who court trouble by choosing names thoughtlessly (such as the famous *foo*), most feature names suggest a specific intent, so a name clash with an inherited feature always justifies going back to the class to see whether it shouldn't be an overriding after all. If you have a feature called *append* in both the parent and the heir, they

probably have related semantics. Language conventions such as overloading, which exorcises name clashes as soon as the signatures differ by as little as one argument type, and "version management" techniques which legitimizes dubious cases by interpreting them as "**new**" declarations, are dangerous. For the debatable sake of short-term convenience, they open the way to serious long-term trouble.

SEMANTIC OVERLOADING

The mention of polymorphism and dynamic binding helps explain where the fundamental contradiction lies. The form of overloading discussed so far may be called **syntactic** overloading; it gives the same name to different operations. There is no good reason to do this. With OT, we can give the same name to different variants of the *same operation*. For example, class *POINT* could be part of a hierarchy of graphical classes that all have a variant of the procedure

move_horizontally (*h: REAL*) -- Displace figure by horizontal offset of *h*. **ensure** -- For all points p in figure, *p.x* = **old** *p.x* + *h*

Then a call of the form

my_figure.move_horizontally (1.0)

will trigger a different procedure depending on the exact type of the object to which *my_figure* happens to be attached at the time of execution of the call. These procedures are all different, and yet they are all implementations of the same basic contract: move all the points of a figure horizontally by *h*. This may be called **semantic** overloading and provides some of the principal expressive power of OO development, allowing various components of a system to know as little as possible about each other, and hence supporting flexible, extendible software architectures.

Syntactic overloading gives the same name to **different operations.** In contrast, semantic overloading—to continue using this term for a while, although of course "polymorphism and dynamic binding" is more precise—guarantees that features from a single seed, appearing in descendants of a common ancestor, **are alternative implementations of the same underlying abstract operation**, such as "move horizontally by a given displacement." This property is not just hand waving: in Eiffel, it is guaranteed by the inheritance rules of Design by Contract, since you can specify a precondition and postcondition at the highest level, defining the common semantics, which the language rules automatically enforce on all the alternative variants. Here, OT brings one of its major contributions to both programming techniques and software architecture.

A MATHEMATICAL NOTE

As a conclusion, let me sketch the mathematical underpinning for the preceding discussion. This presentation is tentative and by no means formal; it just shows the general direction of a satisfactory mathematical model.

If we apply the No-Overloading Principle, the basic mathematical model for classes is simple: we consider a class as essentially a function

where A + B denotes the set of finite functions from A to B. You may view a finite function as simply a set of pairs, such as

$$\{[0, 0], [1, 1], [2, 4], [3, 9], [4, 16]\}$$

(part of the "square" function on integers); it's finite—that is all we deal with using computers—meaning it only has a finite set of pairs, and it is a function, meaning that no two pairs start with the same element. For example, {[0, 0], [0, 1], [2, 4]} is not a function because two of its pairs start with the same element 0.

Our example class POINT gives the function

{["x", horizontal_coordinate_feature],
["y", vertical_coordinate_feature],
["move", move_feature],
...}

where "*move*" is a string (a feature name), *move_feature* denotes the associated feature, and so on.

One of the nice things about such a model—based on semantic work done originally by Luca Cardelli in the 1980's—is that it immediately generalizes to objects. The instances of class *POINT* are, in this model, functions in

where the only names to be considered here are those of attributes, such as *x* and *y* in our example—a subset of the feature names for the class—and *Value* describes the set of possible field values. For example an object of type *POINT* representing the point of coordinates x = 0 and y = 1 is, formally, the function

$$\{[x, 0], [y, 1]\}$$

Returning to classes, the use of a function in this model automatically applies the No-Overloading Principle without further ado. Any form of syntactic overloading would lead to great mathematical complication.

Inheritance also fits nicely in this framework. Since we view classes as functions, a special case of sets, the basic idea for modeling inheritance is that a class is the "union" of its own definition and those of its parent or parents. For example, if *B* inherits from *A*, defined as {["x", feature1], {["y", feature2]}, and itself introduces feature3 with name *z*, then it formally is {["x", feature1],

{["y", *feature2*], ["*z*", *feature3*]}. But when there is a name clash this does not work any more, as the union of two functions is not always a function: with

$$f = \{[0, 1], [1, 2]\}$$
$$g = \{[0, 0], [1, 1]\}$$

both *f* and *g* are functions, but their union $\{[0, 1], [1, 2], [0, 0], [1, 1]\}$ is not since it has two pairs (the first and the third) starting with 0 and giving conflicting values. For functions representing classes, this is the case of a name clash, for example between inherited and new features.

To make a form of union work between functions, and always yield a function, we may use an "**overriding union**" operator that in the case of a conflict always selects the value from the second operand. (This is arbitrary; we could have chosen the first operand instead. Either way, we have to be consistent.) A possible symbol for this operator is \bigcirc , leaning toward the "dominant" operand, in the same way that the bot bar of the Russian Orthodox cross leans toward the thief performed. With this operator, the result is always a function; for example:

 $\{[0, 1], [1, 2]\} \cup \{[0, 0], [1, 1]\} = \{[1, 2], [0, 0], [1, 1]\}$

with the pair [0,0] from the second operand overriding the pair [0, 1] from the first operand. If a class *B* inherits from a class *A*, the resulting function will be $A \, \hookrightarrow \, B$, accounting for any feature redefinition in *B*, which overrides the original from *A*.

In the case of multiple inheritance, it would be a mistake to use overriding union to favor one parent over the other; this would mean that the order in which a class lists its parents has semantic consequences, certainly something we don't want. Instead we resort to the other technique for performing the union of two functions and guaranteeing that the result is still a function: making sure that their domains are disjoint, in other words that no two pairs have the same first element—no two features have the same name. Renaming achieves this by yielding a set of final names that satisfies the No-Overloading Principle. Then we use overriding union with the features of the new class to take into account any necessary feature redefinitions.

This is only the start of a proper formal semantics for classes, objects, and inheritance, but seems to indicate that by sticking to the No-Overloading Principle we can in the end obtain a clean and understandable mathematical model.

DON'T SHOOT YOURSELF IN THE FOOT

Mathematical model or not, this discussion will, I hope, have convinced you that overloading does not go well with OT, and that keeping things simple, in particular the correspondence between names and features, is the key to building a solid basis for the powerful techniques of OO analysis, design, and programming. ■

REFERENCES

1. Meyer, Bertrand. Eiffel: The Language, Prentice Hall, Englewood Cliffs, NY., 1992.