

Wishes for object-oriented languages

Invited paper at LMO 2005, Bern, 9 March 2005

Markku Sakkinen
Department of Computer Science and Information Systems
P.O.Box 35 (Agora)
FIN-40014 University of Jyväskylä
Finland
sakkinen@cs.jyu.fi, sakkinenm@acm.org

Keywords: object-oriented languages, inheritance, multiple inheritance, composite objects, genericity.

1 Introduction

For a long time, I have liked to compare the parameter-passing modes of Algol 60 (some of you are probably old enough to remember) with speakers at scientific conferences. Namely, call by value vs. call by name in Algol, and authors of regular papers vs. invited speakers at conferences. This time I am lucky enough to be in the latter category; I also used to say that these are sometimes people who can no more make enough new contribution to have good chances with reviewers and programme committees.

After this session, we will listen for about two full days to sophisticated technical presentations that will require the utmost concentration to be well understood. In contrast, some lighter entertainment may be quite appropriate in an invited talk. One reason for the lightness is that I could not allocate as much time for preparing the talk as I would have wished. This distributed printed version of the talk does not even contain complete bibliographic references. I hope my Finglish will not be too hard to be understood by speakers of Frenglish.

Today, the emphasis of research and development in object-oriented software engineering has moved from classes, methods and other base-level entities to components, frameworks and other larger entities. Consequently, interests have also moved from programming languages to environments and tools for manipulating these entities. This evolution is natural and welcome, but it does not mean that the existing object-oriented programming languages are already perfect and no further progress on that level is needed. I will present several things that I would wish to be improved in current languages or their successors.

Very little of what I am going to say will be really new, but some points are probably not so well known, or are often forgotten. I have mentioned many of them in my course on OOP at the university (if you want to have my lecture notes in Finnish, feel free to ask). In my papers, I have liked to write about the darker side of things, especially C++, but this time I am speaking about wishes; of course, it is essentially the same approach, but labelled in a more positive and polite way. I will not be as satirical as Bertrand Meyer in his pamphlet “UML, the positive spin” — which I hope many of you have seen.

Several of my wishes are controversial, and I will be happy if they stimulate discussion. Very likely, many of the wishes have also been fulfilled in some OO languages that I don't know, at least not well enough to have noticed. Please comment on such items, too.

My research interests have always been mainly in statically typed languages (STOOPs). This is therefore assumed in most parts of the talk; however, there are things that are rele-

vant also to languages without static typing. Note that I avoid the mistake of calling these “untyped languages” — for instance, Smalltalk is more strongly typed than C++, although dynamically.

I have not been involved in the design of any language, so I have no vested interests of my own, but am equally free to praise or criticise everything and everybody. Some people have recommended that a language designer should not try to invent completely new things, because building a good and consistent language out of well-known constructs is already difficult enough. Very many new ideas and mechanisms have been suggested over the years — just look at a few recent ECOOP and OOPSLA proceedings. Many of them are apparently good and useful, but not too many can be selected into a single language without making it hopelessly complex.

I have not been involved in language implementation, either. This implies that some of my desired features may be so difficult to realise in a language, or tend to make programs so inefficient, that language designers and implementers have had good reasons to avoid them.

2 Modules and nesting

There are good reasons to have a module construct also in an OOPL in addition to classes, as argued e.g. in an ECOOP'92 paper by Clemens Szyperski. Several OO languages do have it, e.g. Modula-3 (of course), Borland's Object Pascal and Ada 95. It was a big disappointment for me to notice that a *package* in Java is not a module in the same strong sense as in the above languages.

Another choice is to allow a class to function also as a module, so that it can contain definitions of inner classes, among other things. This was the original principle in Simula, and very much exploited there. Nested classes are possible also in C++, in a similar meaning. I think it is a drawback of Eiffel that this possibility is missing; but there are arguments in favour of that decision.

In Java, inner classes now come in two main flavours, *static* (a term I don't particularly like) and non-static. The previous ones correspond to Simula, while the latter are more interesting: every instance of an inner class belongs to an instance of the outer class and has access to the features of that object.

In BETA, inner classes have an even stronger meaning: they are actually metaclasses in the sense that each object of the outer class has an inner class of its own. This allows some real-world situations to be modelled more directly. For instance, one can define a class `Car_model` with an inner class `Car`: clearly, every car is an instance of some specific car model.

One important difference between modules and nested classes is that inner classes of the same enclosing class have no special access rights to each other. In Java, however, they have full access to all features of the *enclosing* class. In most modular languages, there can be no information hiding within a module. I think that it is best if the visibility within a module can be chosen for each feature, as in Java (see 3.1).

Simula retained the possibility of *nested subroutines* from Algol, and it is useful for abstraction and information hiding. However, most current OOPLs do not allow it, although it would not be difficult for programmers to understand nor for implementers to realise. It would be particularly convenient in those languages in which method bodies can be written separately, such as C++: inner subroutines (methods) would not need to clutter class definitions at all.

3 Inheritance

It can be argued whether inheritance is the most important characteristic of object-oriented languages, but at least it is their most conspicuous characteristic. I will first present some viewpoints and wishes that are relevant already for single inheritance (SI), and then discuss problems that appear only with multiple inheritance. Most ideas in the first two subsections originally come from my paper in Computing Systems already in 1992; I assume that many of you have never seen it. The last three subsections briefly discuss some newer issues.

3.1 Single inheritance

Different OOPs have different sets of *visibility levels* for the features of classes — also different syntactic mechanisms for denoting them, but that is less important. In C++, not actually the visibility but only the *access level* can be controlled, so some features that cannot be used can nevertheless cause some harm.

Most STOOPs have at least the alternatives *public* and *protected* (visible only to the class itself and its descendants). In my opinion, the *private* alternative, which exists e.g. in C++ and Java, is good and useful. However, there are good reasons also for the decision in Eiffel not to offer that choice. On the other hand, Eiffel has an intermediate form between public and protected: allowing access only to explicitly listed other classes.

In a language with modules, accessibility becomes a little more complicated. A private feature can in principle either be visible within the module or not. In Java, the former is the default visibility and *private* means the latter. A protected feature has three choices in principle: visible also within the module, visible only to descendants, or visible only to descendants in the same module. Java has only the first possibility, which I consider a mistake (especially because the packages of Java are not closed entities). C# offers also the second alternative, but not the third, which would also be quite useful sometimes.

At one point, the meaning of *protected* in C++ was changed from the original, simple and easily understandable one. Now, a method of a class *S* can access an inherited protected feature *F* only in objects that are statically known to be of class *S* or a descendant. The rationale for this restriction is very weak; a feature that needs such protection should be defined in *S* and not in an ancestor in the first place. Its anomalous nature is accentuated when the protected feature *F* is defined in a class whose all methods are abstract (deferred): there cannot exist any method that can access *F* in all objects that have it.

The above restriction can be a serious hindrance to programming. Although intended to strengthen the protection of features, it can sometimes have the opposite effect: features must be declared public instead of protected. The same principle has unfortunately been adopted also in Java.

Instead of such a restriction on protected features, a sensible thing would be to allow *per-object visibility* to be defined for private and protected features, i.e. that some feature could be accessed only in the current object. This is indeed possible in Eiffel by explicitly not exporting that feature to its own class. It has seemed strange to me that such a feature can be exported to *other* classes, but that can be useful in some special cases, such as test classes.

One specialty of C++ would be worth adopting to other languages in my opinion, although it adds one dimension of complexity to the access rights. Namely, the same three access levels apply also to superclasses (or, I would say, to inheritance relations). Public inheritance corresponds to inheritance in almost all other languages; but the default in C++ is private. Private inheritance is very useful when the intention is to reuse implementation only without any *is-a* relationship (or subtyping). If a class *S* inherits a superclass *R* privately, *S* objects can be handled as *R* objects only in the methods of *S* itself (and friends). The difference between the access levels of superclasses becomes especially interesting in

multiple inheritance.

Finally, a further design flaw in C++ (in my opinion) is that a private virtual method can be redefined in a subclass, although the inherited method cannot be invoked. Normally, one would not declare any virtual method as private, but an originally public or protected method becomes private in a class that inherits it from a private superclass. The situation then becomes anomalous for the descendants of this class.

3.2 Multiple inheritance

According to a well-known saying by Alan Snyder, multiple inheritance (MI) is a good thing, but nobody knows how to do it right. I still believe that it *can* be done right, but do not know of any existing language in which it *has* been done right. As you know, no new mainstream OOP with full MI has appeared for a long time, nor has any originally SI language recently been refurbished with MI in its evolution. The latter is what happened to C++ long ago, and the operation was not completely successful although the patient survived. Eiffel and C++ have remained as the two primary exponents of MI among well-known statically typed languages for over 15 years.

It is important to distinguish between (in the terms of my ECOOP'89 paper) *independent multiple inheritance* (IMI) and *fork-join inheritance* (FJI). More precisely, we can speak of independent *ancestors* of a class on one hand, and fork-join or *multiply inherited* ancestors on the other hand. The latter are inherited through more than one inheritance path; in Eiffel a class can be multiply inherited even directly.

In my opinion, one basic requirement for MI is *symmetry*, i.e., that all superclasses of a class should be on a completely equal standing. This seems necessary at least when inheritance is expected to mean an *is-a* or *is-a-kind-of* relationship. Both C++ and Eiffel fulfill this requirement in most cases. In Eiffel, however, intermediate classes on different inheritance paths from the same multiply inherited ancestor cannot be treated symmetrically when feature replication (see below) happens.

The symmetry requirement is obviously controversial: at least in the Flavors-CLOS approach to inheritance, the order between superclasses is significant. Among others, many French researchers have studied different linearisation principles and algorithms to improve this approach. On the the other hand, Henry G. Baker has shown that linearisation can have truly pathologic and counterintuitive consequences in some sufficiently complex inheritance structures.

I have divided the multiple-inheritance approaches of OOPs into two opposite camps: *subobject-oriented* and *attribute-oriented*. The first camp includes C++ and Smalltalk: in these languages an object can always be considered to contain a subobject corresponding to each ancestor class of its exact class. The second camp includes most LISP-based OOPs and the well-known Cardelli theory: attributes inherited from different ancestors are unified if they happen to have the same name. In this approach, completely unrelated classes can have unexpected interactions with each other in the context of a common descendant, through the unified attributes. In my opinion, the subobject-oriented approach is therefore the clearly better one.

Eiffel actually lies between the two extreme camps. In an IMI situation, it is almost subobject-oriented: attributes originating from different ancestors can never be unified, but their types can be covariantly changed, so a subobject need not be exactly similar to a direct instance of the ancestor. However, in an FJI situation it is attribute-oriented: each attribute can be duplicated or not, independently from others (duplicated if renamed in some branch).

Methods are treated differently from attributes in some languages. Thus, Eiffel allows independently inherited methods to be unified if the programmer wants, by renaming if their names differ. In C++, such unification happens automatically and cannot be prevented if virtual methods from different superclasses happen to have the same signature.

In my opinion this is a mistake, and it has been copied to Java. This is almost the only MI-related mistake that was *possible* to make in Java, because MI is allowed only from interfaces. The problem has been solved in C# in the simple way I had proposed: qualification by interface name in an implementing class if the methods are to be kept separate.

Still more important than symmetry between superclasses is that there should always be a *one-to-one correspondence* between ancestor and descendant (sub)objects. To be precise: if class A is an *accessible* ancestor of class D, there must be exactly one accessible A subobject in every D object. However, when the D object itself occurs as a subobject of a further descendant that multiply inherits A, the A subobject will be shared with subobjects of other intermediate classes.

In most OOPs, all ancestor classes are accessible to all descendants. However, in C++ A is not accessible to D if there is some intermediate class in the inheritance chain that inherits A (or a descendant of it) *privately* (see 3.1). Briefly, accessible subobjects should always be shared, and inaccessible subobjects always replicated.

Unfortunately, in C++ A subobjects are shared between those subclass whose inheritance from A has been declared `virtual`, and other subclasses have replicated subobjects. Thus, this is completely independent from accessibility. The C++ principle can lead to two kinds of anomaly: a D object can have several A subobjects, and several D objects can share a common A subobject. Both situations are inconsistent with the “is-a” relation that public and also protected inheritance should imply.

3.3 Reverse inheritance (“exheritance”)

In all current OOPs, inheritance only allows new classes to be defined as *specialisations* (in a wide sense) of existing ones. However, in OO modelling and design and in the evolution of existing software, it is quite natural to apply also *generalisation* when one observes that some originally independent classes have sufficient commonality. Because there are no language mechanisms for that purpose, generalisation requires the definitions of those existing classes to be modified.

There is some literature speaking of “OR-inheritance”, which means ordinary inheritance, and “AND-inheritance”, which means generalisation. Claus Pedersen suggested in an OOPSLA 1989 paper that direct support for generalisation should be added to common OOPs. I warmed up the idea in a short paper at the ECOOP 2002 Inheritance Workshop, calling it “exheritance”, but a better although longer term could be *reverse inheritance* (RI). It appeared that Pedersen’s original suggestion had one serious flaw, but it could be corrected.

Reverse inheritance can be selective, i.e., not all common features need be exherited. It is also quite meaningful to base such selective RI on *one* existing subclass, so that further subclasses can later be derived from the new superclass. As one could expect, RI looks quite easy and simple if only common parts of the *interfaces* of classes are “exherited”. Exheriting also implementation is more tricky, and Pedersen’s flaw was here. The addition of reverse inheritance (RI) to an existing language would seem to be quite feasible.

Reverse inheritance would make it possible even in the subobject-oriented approach to unify corresponding features from two or several originally independent ancestors in a common descendant, thus narrowing the gap to attribute-oriented inheritance. It would suffice to define a new generalisation class that includes those features. A possibility of renaming would be quite useful in such cases, because “the same” feature may very well be named differently in two classes that have been designed independently of each other. This would correspond to what is already possible in Eiffel for methods in ordinary inheritance (cf. 3.2).

One key point in my suggestion is that RI should in no way affect the subclasses from which new superclasses are exherited, similarly as ordinary inheritance does not affect the superclasses in any way. Further, if a class is modified, that may affect all superclasses

exherited from it, just like all subclasses inherited from it. However, the modification affects only those superclasses that have exherited at least one feature that gets modified.

Philippe Lahire and others at the University of Nice – Sophia Antipolis have worked on a different approach, in which all definitions in an exherited superclass are inherited to the subclass, just as from an ordinary superclass. We hope to research these issues together in the near future, when I will make a one-month visit to Sophia Antipolis.

3.4 A tricky example

One challenging example of fork-join inheritance for me has been the “intercontinental drivers” from Bertrand Meyer’s *Object-Oriented Software Construction*. The basic idea is that the class `Driver` has two subclasses, `French_driver` and `US_driver`, and these have the common subclass `French_US_driver`. In this class, some attributes originating from `Driver` appear once, but some others are duplicated.

I originally criticised (at ECOOP’89) the fact that such a class structure is possible in Eiffel, for the reason that inheritance does not preserve subobjects. That problem could be solved in the subobject-oriented approach by reverse inheritance if one would not want to modify the existing class `Driver` (it might be a library class used also in many other places): moving the shared attributes to a new superclass, say `Basic_driver`. In C++, `Driver` would inherit it “virtually”.

The more severe problem still remains that there are two `Driver` subobjects sharing one `Basic_driver` subobject and contained in one `French_US_driver` object, thus no one-to-one correspondence. From my viewpoint, this means that one tries to use inheritance more than is appropriate. It would be more appropriate to use a 1-N *association* (see 4) between `Driver` and `Basic_driver`. One would need some additional programming, but the class `French_US_driver` might be disposed of.

To handle such situations easier, many researchers have proposed *roles* to be added to OOPs. I have not studied these proposals recently, but I think that a role mechanism could be a worthwhile addition to a STOOPL. In this case, one or more `Driver` roles could be attached to a `Basic_driver` object.

3.5 Mixins

One simple way to look at possible mixins in STOOPLs is the following, which does not cover everything mixins can do in some languages.

In standard inheritance, the new and modified features of a subclass are defined on the basis of a fixed superclass. In a mixin, the increment is defined independently and can be combined with different superclasses to create subclasses. These superclasses can be defined after the mixin, but the mixin definition must contain requirements for allowable superclasses: there was e.g. an OOPSLA’93 paper by Franz Hauck on this subject. Most importantly, some methods can be required to exist in the superclass, just as abstract (deferred) methods are required to be ultimately implemented (effected) in a descendant class.

Using mixins can be regarded as an intermediate between single and multiple inheritance. Unlike in true MI, it is natural that the order in which different mixins are applied *is* significant. It seems also straightforward that two mixins can be combined; the supermixin may then offer all or part of the features required by the submixin.

It is perhaps both the main strength and the main weakness of mixins that they are written independently of the superclasses to which they will be applied. However, in some languages mixins must be defined for particular ancestor classes; the mixin programmer will then know in what context the mixin methods will operate. It is obviously a good idea that the applicability of a mixin *can* be restricted to some already defined superclasses.

4 Composite objects

Object orientation has often been advertised to be good and natural for the handling of composite (or complex) objects. In reality, almost all current OOPs are weak in their support of composite objects. Some OO *database* languages and UML are far better on this point.

Attributes and other variables in most OOPs always have reference semantics, which means that physically embedded part objects are not possible. That is not a severe problem yet, but no mainstream language offers a distinction between ordinary references and those intended to point to a logical part object. In the ORION OODBMS the latter were known as *composite links*. It depends thus fully on the programmers to take care of what is supposed to constitute a complex object. Among other things, copying and comparing become very complicated and tricky because of this, as explained by Peter Grogono and myself in a paper at ECOOP 2000.

Another problem concerns also those languages that allow physical composite objects, such as C++. Namely, given a reference to a part object, one cannot get to the composite — and not even know whether it is a part or an independent object. There has been a simple solution to this problem in an experimental version of BETA, but for some reason it has not been incorporated into the standard language (as far as I know). Every object has an automatically defined attribute *location*, which either points to the enclosing composite object, or is a null reference in an independent object. I would recommend this mechanism for all languages. However, it causes extra overhead, and so it should be optional for each class.

With Java's non-static inner classes we have an inverse situation: instances of the inner class have an implicit pointer to the instance of the enclosing class through which they have been created, but not vice versa. It is possible to keep the collection of inner-class objects in the outer-class object by programming all constructors of the inner class suitably. However, this will prevent all inner-class objects from being garbage collected as long as the outer-class object exists.

Some languages have supported *weak references*, which do not prevent garbage collection. This would be a nice feature for some situations like the above. However, it is not such an essential requirement for complex objects as composite links (or composite references).

A more general solution that would cover also “back links” from parts to composites would be support for bidirectional associations, both 1-1, 1-N and M-N. These are familiar from UML and its predecessors, but also the ODMG (unofficial) standard for object databases prescribes them. Of course, associations (especially M-N) are rather heavy constructs, so even if a language supported them, it should be possible to have also ordinary references between objects.

Bidirectional associations would solve some common problems with conventional references. The worst problem is the danger of dangling references in languages like C++ where objects are explicitly deleted. The converse problem appears in languages with garbage collection: if one would want to have a certain object deleted, one should know all references to it from other objects and remove them; but that is impossible in general.

5 Value-oriented vs. variable-oriented types

Bruce MacLennan argued in an article in SIGPLAN Notices in 1982 why programming languages should support both values and objects (in a wide sense) and a clear distinction between these. I am using the term “variable-oriented” here as the opposite of “value-oriented”.

Most OOPs don't support value-oriented classes very well, in contrast to Ada and C++. Such a class should have at least assignment and equality comparison available. Reference

assignment is equivalent to value assignment only if all objects of the class are immutable. That requires that no methods can cause side effects to objects of the class.

Most OOPs don't support constant objects of normally mutable classes either, again in contrast to Ada and C++. With reference semantics one would actually need even a concept of *deep immutability*, i.e., that also all logical part objects (direct and indirect) are immutable. With physical part objects that comes automatically.

Enumerated types are the most common kind of user-defined *simple*, value-oriented datatypes. Although they are a very convenient abstraction mechanism and easy both for programmers and language implementers, most OOPs don't support them — do I need to say that Ada and C++ are among the positive exceptions even in this respect? Enumerations have now been added to Java in version 1.5, but I was quite surprised to learn that they were made into fully-fledged classes. That brings some advantages, but Java enumerations cannot be used in *switch* (case) statements, one of their traditional main uses.

6 Genericity

(I had intended to write a lot more on this topic!)

Ada was not the very first programming language with genericity, but the first widely used one. Its principles and mechanisms seem to have stood the test of time well, so they are a good point of comparison for newer languages.

Among the earliest STOOPLs after Simula itself, at least Trellis/OWL and Eiffel supported generic classes from the beginning, and genericity was later added also to C++. I was probably not the only one who considered its lack to be the most significant (and perhaps most surprising) shortcoming of Java. Genericity has now finally been added to Java 1.5.

In Eiffel, genericity can be said to be *semantic*, like in Ada: already a generic unit can be understood as such, and to some extent also compiled, at least in principle. The constraints on a formal generic parameter determine in both languages how it can be used within the unit; in Eiffel only classes are possible generic units, for obvious reasons. Eiffel is more restricted than Ada in the sense that constraints can be expressed only as inheritance relations (upper limits). This implies among others that at least the constraint classes must be defined before the generic class is written; the Ada principle is more open-ended.

In C++, genericity (*templates*) is purely syntactic: not much more than lexical analysis can be performed on a template itself, and compilation can be done only for each generic instantiation. It is therefore logical that no constraints can be specified. There are several tricky things about C++ templates, and they turned up to be much more versatile and powerful than had originally been intended. This has led to so-called *template metaprogramming*, but I think C++ templates are difficult and dangerous to program for ordinary uses. Assembly language is also very versatile and powerful!

Java's new genericity principles are rather similar to those of Eiffel, and can be seen as the extreme opposite to C++. Generic classes are completely compiled as such, and nothing of instantiated classes exists at run time. This is a drawback in some situations: e.g., the element type of a container cannot be queried at run time (standard container classes are now generic, of course).

Another drawback in Java is that primitive types cannot be used as actual generic parameters; in Eiffel they can because they are expanded classes. It is surprisingly difficult to define generic classes with arithmetic types as generic parameters, e.g., for vectors and matrices. Namely, the *wrapper classes* corresponding to primitive types have no arithmetic operations, and one cannot even define subclasses to add those operations, because the wrapper classes have been declared `final`.

In Ada, each instantiation of a generic unit *must* be named, and inventing new names can obviously be a nuisance sometimes. On the other hand, instantiations *cannot* be named

in Eiffel and Java, but they must always be denoted with the full actual parameter list. This can probably be a much worse nuisance in some cases. In C++, an alias *can* be given to a generic instantiation when desired by using `typedef`, like for any other type.

In Eiffel, if class B is a descendant of class A, then for any applicable generic class G, G[B] is regarded as a descendant (subtype) of G[A]. This is questionable because it is not statically type-safe; but Eiffel also allows covariant redefinitions of the types of class attributes and method parameters, which are similarly unsafe. Of course, type errors are caught at run time. This subtyping relationship is *not* assumed in C++ and Java.

7 Miscellaneous

There are some other interesting aspects that I had no time to discuss in this paper, at least persistence, exceptions and transactions. It has seemed to me over the years that there are too few contacts between OO programming language people and OO database people, and that the well-known “impedance mismatch” between programs and databases seems to remain strong. Both parties would have a lot to learn from each other. Kazimierz Subieta’s *stack-based approach* to an integrated programming and database language looked like a promising beginning about ten years ago, but there has not been much progress on it later.

Concurrency (or parallelism) is one aspect in which object orientation would appear very natural. A number of special concurrent OOPLs have been described in the literature and also implemented since the 1980s, but they have remained niche languages at best. Mainstream OOPLs have either ignored concurrency completely, e.g. C++, or not been able to integrate it smoothly with objects.

Problems in the concurrency principles of OOPLs have already been analysed by real experts in that area. In an article in SIGPLAN Notices in 1999, Per Brinch Hansen concluded that Java ignores the last twenty-five years of research in parallel programming languages!