Disciplined Inheritance

M. SAKKINEN *

University of Jyväskylä, Department of Computer Science

ABSTRACT

Several other properties of objects are regarded as more important than inheritance. Inheritance is classified into two alternatives: incidental and essential. Incidental inheritance is a matter of implementation and software engineering, and should be completely encapsulated. Essential inheritance models important relationships in the problem domain, and should be completely visible. It is then attempted to model inheritance in terms of aggregation and dependence. A number of omissions and divergences and some surprising parallels are found in the treatment of inheritance in existing object-oriented languages and previous literature. This contribution does not pretend to close the case yet: for instance, intermediate forms between purely incidental and purely essential inheritance are not discussed.

Keywords: Inheritance, encapsulation, complex objects, type compatibility, delegation, programming discipline, class design.

1 INTRODUCTION

Many of us can agree with Bertrand Meyer when he states in the Preface of [Meye2], about the design of the Eiffel™ language:

> "The foremost influence has been that of Simula, which introduced most of the concepts twenty years ago, and had most of them right; Tony Hoare's remark about Algol 60 — that it was such an improvement over most of its successors — applies to Simula as well."

The first definition of SIMULA 67, later renamed simply Simula™, is [DaNyMy]. Single inheritance is well defined already there, although the word used is 'concatenation'. The

---

* Electronic mail address: markku@jytko.jyu.fi (Bitnet: SAKKINEN@FINJYU).
Postal address of department: Seminaarinkatu 15, SF-40100 Jyväskylä, Finland.

language definition even in general has a down-to-earth style. Anthropomorphisms and overstatements have become part of object-oriented parlance later, apparently stemming from the Smalltalk (!) community. Examples of overstatement are 'active objects' and 'message passing' as applied to Smalltalk-80™ [GolRo] and similar languages. As [YokTo] explains, reconciling true asynchronous message passing with Smalltalk "message passing" is not quite simple.

'Inheritance' is an appealing word because its meaning in object-oriented programming (OOP) is so analogous to its usual meaning, which in turn is familiar to everybody. Still, probably because of this intuitiveness, there seems to be no common definition of 'inheritance'. Even in the ordinary meaning, there are many different kinds of inheritance, at least biological, juridical, and cultural; each of these has different rules.

To advance the state of OOP, we will need on one hand systems [AghHe, HaiNg, MinRo] that give the programmer more explicit choice over inheritance and other mechanisms that are fixed more or less ad hoc in conventional languages, and on the other hand tentative rules and restrictions that promote "better" programming [JohFo, LiHoRi, Sakk2, LieHo]. Here we stress the latter side of the coin, i.e. disciplined programming and software engineering. Features aiming at "exploratory programming" need not necessarily make the programmer into a Vasco da Gama or an Amundsen; she may well become Alice in Wonderland, never knowing what metamorphoses some seemingly innocent act may cause.

The main purpose of this paper is to suggest how the problems and ambiguities of inheritance could be controlled, and how inheritance could be modelled by other mechanisms. As I currently, agreeing with [Amer1,2], do not believe that inheritance is *the* central principle of OOP, it seems obligatory to present first the framework in which it will be discussed. In conscious provocation, I will list several properties of objects as more fundamental than either inheritance or *classes* — although I *do* believe in classes. We will also try to classify the intents of inheritance in a dichotomy.

The great variability of object-oriented terminology makes it problematic to write "cross-language" articles. Here we will not even try to speak about each specific language in the terms of its own literature, but neither will we succeed to establish a completely language-independent one-to-one relationship between terms and meanings. I apologise for the lack of concrete examples; they would have taken too much space.

2 WHAT ARE OBJECTS

Here are again one person's views on what are the more important and less important characteristics of things to be called *objects*. Necessarily, most of these are requirements on the *system* (programming language or other) that manages objects.

Smalltalk-80 is a trademark of ParcPlace Systems.

The most important property of an object is **identity**. The identity must be unique within a given universe of objects and a permanent property of each object. The most degenerated possible kind of objects are those that have no other properties at all, but even they can be interesting — the whole general set theory is built on them.

The second property in importance is **integrity**. By object integrity I mean that no property of an object must be changed except by operations that *intend* and *have the right* to modify that object. The meaning of 'intend' here is very wide, covering even extremely indirect effects. — Weak support of object integrity in a language typically goes together with the lack of explicit identity: C++ [Stro1, Sakk1] is a good example.

The third property is that objects may in general be **created** and/or **deleted**. However, there can be objects that (at least conceptually) need not be created and/or cannot be deleted. The *possibility* of deletion can be a property of individual objects, and in that case need not be a permanent property.

The fourth property is that an object has a **type**. The type can be primitive; or it may be defined either *constructively* (structurally) from other types or *behaviourally* (as an abstract datatype), or in a mixed way. In conventional OO systems the type of each object is permanent, but sometimes there are good reasons for letting even the type change [SkaZd].

One aspect of type is that an object may have **attributes** (data components). Attributes can be either non-object values or themselves objects. A special case of a non-object value is a *reference* (or surrogate), which is equal to the identity of some object or **nil**. In almost all OO languages and in some languages not considered object-oriented, e.g. CLU [Lisk&al], all *variables* are of a reference type. This has had consequences for the concept of inheritance (cf. §5). Some languages employ *capabilities* [AnPoWa], which combine references with access right information.

A second aspect of type is that an object may have **operations** (*methods*) that are executable. The difference between attributes and operations can become blurred in systems with triggers or access-based programming: what seems simply a read or update access to an attribute may actually cause the invocation of an operation.

A third aspect of type is that an object may have **processes** (*activities*). This is impossible in most current object-oriented languages: either they have only one sequential thread of control, or a process is a special kind of object or a separate implementation notion. In Simula, BETA [BKri&al], Actor languages, and newer concurrent/parallel OOPL's [YonTo], objects form also the units of concurrency (true or quasi-). In Emerald [Blac&al], operation invocations may proceed concurrently with each other and with their object's process. The Parallel Objects model of [CorLe] allows a high degree of controlled *intra-object concurrency*.

Sometimes [Viha, Sakk3] *abstract entities* such as (constant) integers are not considered to be objects. The argument is that it makes no sense to say that e.g. the integer 743 is "created" or "deleted", and saying that "the integer adds another integer to itself" feels hardly more appetising to a mathematically trained mind. Under this view, abstract entities need not have identities in the same sense as objects. — The opposite viewpoint *is* well motivated in [Lieb2] for the classless Actor languages. In a more typical language like Smalltalk-80, however, it appears hard to define consistently how a class that inherits e.g. from Integer and adds some instance variables should behave.

3 LESS IMPORTANT PROPERTIES OF OBJECTS

Compared with Simula, most current object-oriented languages prohibit effective nesting, at least that of class definitions or their equivalents [BuhZa]. In compensation they enforce a stronger **encapsulation**, often so that only the operations and no attributes nor the process of an object can be directly accessed from outside the object itself. Although this is mainly beneficial, it sometimes requires operations concerning several objects to be defined in an unnaturally asymmetric manner; an exception is CLOS [Bobr&al] with its "multi-methods". Some languages allow more selective encapsulation, e.g. C++ with its **friend** declarations that can solve most of the asymmetry problem. — We will have more to say about encapsulation at several places.

One of the key concepts in OOP is **inheritance**. However, in spite of its flexibility and other attractions, unrestricted inheritance has been found to cause many problems as well. One fundamental contradiction is between inheriting specifications or behaviour on the one hand, and implementation or structure on the other hand. Also, single inheritance has lead to many unnaturally asymmetric constructions; this has later been remedied by multiple inheritance in many languages, but multiple inheritance has caused new problems (name conflicts etc.). We will base our constructions on multiple inheritance throughout. — Sometimes the concept of *delegation* [Lieb1] is proposed instead of or as a complement to inheritance [AghHe]. The claim that delegation is more powerful than inheritance was refuted by [Stei1], however. Note that some authors, e.g. [UngSm, HaiNg], use 'inheritance' approximately in the same meaning as some others use 'delegation'.

One feature connected to inheritance that is invariably counted as a principal factor in the flexibility of OOPL's is the redefinition of inherited operations. Nevertheless, this possibility is also a principal source of devious programming errors. We will suggest restrictions to such redefinitions in sections 6 and 7.

There can be **existential dependences** between objects; the attitudes of typical OO *programming* languages (garbage collection of totally unreachable objects) and most *database* systems (explicit, possibly cascaded deletions) are opposite to each other. There are some recent attempts [Kim&al, Sakk3] to combine these views. Existential dependences are a very special case of *constraints* in an object system. They can be efficiently enforceable, whereas more general systems of constraints very easily become undecidable. Dependences will be used in this paper as an auxiliary tool to model or even replace inheritance

with aggregation. — The assertions of Eiffel seem to be an example of efficiently manageable constraints in an OOPL, but they are viewed mainly as a specification and debugging device. In ThingLab [Born] constraints are the essence of the system.

Most OOPL's are built around **classes**. A class is more than just a type: it is also a first-class object in e.g. Smalltalk and an "almost object" in e.g. C++ (a C++ class can have **static** variables that are accessible to all its instances). Typically, all operations are defined within classes and cannot be redefined in instance objects, although they are called "instance methods" in Smalltalk and many other languages. As an inheritance from C, C++ objects *can* have attributes that are pointers to functions, but these are not considered operations of the object. Inheritance also goes strictly between classes: an instance object cannot inherit anything but is instantiated with what the class has inherited.

There are languages that dispense with class as a primitive concept, e.g. Actors [AghHe] and Self [UngSm]. Instantiation is there substituted by replication of a *prototype* or *exemplar* object. This approach often goes together with replacing inheritance by delegation, but not always [LaThPu]. Our presentation will mostly be independent of the class-prototype controversy; I am suggesting elsewhere [Sakk4] that to reject classes is to throw the child out with the wash.

In Algol 68, Ada®, and some other languages, although *types* are only compile-time entities, formal type parameters can be used to define **generic** (or *parametric*) packages, procedures, new types, or whatever (depending on the language). The advantages of genericity are thoroughly discussed in [Meye1,2]. Generic classes have not been implemented in many OOPL's besides Trellis/Owl™ [Scha&al], Mode [Viha], and Eiffel. Of course, the whole principle is relevant only to strongly typed class-based languages; the majority of currently popular languages is weakly typed.

It is not common to regard the type of an object (or of a non-object value) as a first-class value that can e.g. be assigned to a variable of type 'type'. However, this approach has been chosen in the Mode language, and it looks like a sensible enrichment of the type system. It does *not* mean that the type of an object could be magically changed by updating its "type field".

4 ESSENTIAL VS. INCIDENTAL INHERITANCE

There is often *too much* inheritance in object-oriented programming. Programmers can sometimes apply inheritance when plain aggregation would be more suitable, but even some languages as such force too much of a good thing. The Smalltalk tradition that there must be a universal superclass (Object) causes problems, especially if it is still enforced with multiple inheritance. This requirement does not exist in Simula and its direct

---

Ada at least was a registered trademark of the United States Government (AJPO).

followers.

There are also many *flavours* of inheritance. The difference between the views of inheriting specification (behaviour) or implementation was mentioned several times at ECOOP'88 in Oslo (panel discussions are not recorded in [GjeNy]), often saying the former to be typically European and the latter typically American. We propose two new terms that seem to clarify the picture a little at least for the present purposes: **essential** and **incidental** inheritance. Inheritance of implementation only is always incidental. Inheritance of specification is essential, whether implementation is inherited also or not. In the classification of [HaiNg], the case "Code Sharing and Reuse" corresponds to our incidental inheritance, the others to different facets of essential inheritance: "Type Theory Inheritance" implies inheritance of implementation as well, while "External Interface Inheritance" and "Simple Polymorphism" do not. Ian Holland has proposed the following interpretation:

> Incidental inheritance seems to appear as a result of software engineering and program design. Essential inheritance occurs as a result of domain analysis and system design.

This captures the heart of the matter.

We do not claim the distinction between essential and incidental to be absolutely sharp. One might say that as the commonality decreases and the differences increase between two "like" types [WegZd], their relationship becomes more and more incidental. A simple example that could be regarded as incidental inheritance of specification can be drawn from the ever-popular domain of data structures. A stack and a queue can have the same set of operations, and even with the same signatures. Nevertheless, if a software designer should change the operations of a stack, this need not necessarily affect the interface of a queue. There might further be e.g. a 'button' class in the same system, which could also have a 'push' operation like a stack. The connexion between these operations would be a purely incidental name collision — we have not got enough space here to explain how we would completely exclude cases like this from "inheritance" (cf. §9).

Examples of insightful recent papers on essential inheritance are [WegZd] and [LKnu]. On the contrary, the interesting paper [Snyd3] (a slightly modified version of [Snyd2] with a CommonObjects example added) focusses on incidental inheritance. Snyder explicitly uses 'inheritance' in the meaning 'inheritance of implementation' (so does [JohFo]). He does mention inheritance of specification, calling it 'subtyping'. The same terminology is used in [Amer1], a paper that is fully relevant to non-parallel languages as well. Readers should note the difference in terms between these papers and ours. However, we will actually not study situations in which *only* specification is inherited, because there would not be much to say about them (well, [Amer2] says a lot). We will thus stay on the common ground that practically all authors call inheritance.

Trellis is a trademark of Digital Equipment Corporation.

Since we consider the properties of §2 more important than inheritance, we will not accept such features of inheritance that would clearly be in conflict with any of those properties. As pointed out in [Amer1], inheritance will probably play a less important role in parallel than sequential languages. We will not even discuss inheritance of processes, although already Simula offers the **inner** keyword for that purpose. It allows the superclass to define at what point the body (process) of any subclass shall be executed: a kind of converse to the **super** construct of many languages that allows a subclass operation to define that a superclass operation is to be invoked.

5 INHERITANCE AS AGGREGATION

Contrary to almost all non-OO languages, almost all OO languages enforce the principle of totally indirect *aggregation* (composition): variables in objects cannot contain subobjects directly but only references to other objects. The obvious advantage is that classes remain really independent of each other's implementations. Certainly the best-known exception that treats aggregation (both arrays and records) like "ordinary" programming languages is C++. EXTRA [CaDeVa] is similar, but more in Pascal style and handles even sets. Both C++ and EXTRA *allow* a class designer to use pointers (references) whenever more appropriate.

The principle of indirect aggregation must be a major culprit for the overuse of inheritance in OOP. It is also the main reason why we need a concept of incidental inheritance at all, instead of mere aggregation: the parent part normally *is* physically concatenated to the non-inherited part of an object, conforming to the original Simula approach (cf. §1). However, if we add three restriction capabilities that are missing from most languages today, we can model inheritance (both kinds) by aggregation. Aggregation is a much less ambiguous concept than inheritance. We will concentrate on *data* (instance variables) and not speak anything yet about inheriting *operations*. It is actually the more important half of the question, but it depends on the kind of inheritance, and will be discussed in the following sections. — We will speak in terms of classes, but the discussion is applicable to prototype-based languages as well.

The "mathematical difference" of a subclass object and a corresponding superclass object is not defined in the conventional OO view, or in any case it is not an object. This can be seen as a defect in the object or class algebra; the problem has been recognised in [Stei2]. We strive to a model in which the difference will always be an object. We will thus allow even objects that have a "negative compound type", while [Stei2] does not permit such types to be instantiated; but these objects cannot exist alone, only as parts of complex objects. In [WegZd] §7 it is noted that it is often most convenient to define largely similar classes by giving only their incremental differences. This thought is not far from Stein's type expressions.

To be definite, suppose class C has the immediate superclasses $D_1, ..., D_n$. *Let us regard any instance O of class C as a complex object, consisting of a full-fledged object* P_i *of each*

class D_i, and one further object O^- that contains the non-inherited instance variables (components) of O. Let us call the class of O^- analogously C^-. With a slight change to common OOPL usage, we could actually define C^- first without naming the superclasses, and then C as a "sum class". This would then explicitly be "programming by difference". The exact structure of the complex object O turns out to be different in essential and incidental inheritance, and so is deferred to the following sections.

To obtain the usual semantics of inheritance, the three restrictions we must be able to put on every subobject (parent component) P_i of O are as follows:

(1) P_i is deleted when and only when O has been deleted. (In the terms of [Sakk3], P_i is "immediately completely dependent" on O.)

(2) The connexion between O and P_i is permanent.

(3) O must export no reference (surrogate) to P_i neither to its other components nor to its clients (objects that invoke O's operations).

In fact, these restrictions apply to O^- as well; we can denote that subobject also by 'P_0', and C^- by 'D_0'. The inheritance model of conventional OOPL's certainly fulfills these conditions: (1) and (2) because each P_i is physically contained in O, (3) because the components are not objects at all and thus cannot be referred to.

The class C^-, if it really needs a *super*class to be meaningful, is a kind of mirror image of an *abstract class*, which needs a *sub*class to be meaningful. Our present approach implies that even abstract classes can be instantiated, but their instances must always be components of other, subclass objects. We can say that they, too, have negative compound type. Thinking in terms of difference classes seems to make the relationship between subclass and superclass more symmetric. A further advantage of difference classes is that some of them may be sensibly combined with different parent classes, thus reducing duplicated declarations.

The "mix-in" classes (flavors) of Flavors (mentioned in [Snyd3] but not in [Moon]) seem to be largely equivalent to our difference classes. In Flavors, parents are called 'components' as they are here, but the word has a slightly different meaning — instance variables are *not* components. Further, the "mixing" principle means that there will generally be no real subobjects in a Flavors object, because components lose their integrity. In most other OOPL's a parent object essentially exists within each child object, but one cannot refer to it as such (as noted above): it has integrity but no identity.

6 MODELLING INCIDENTAL INHERITANCE

Now we take up the questions of inherited operations and complex object structure. One would expect it to be more straightforward to model or replace incidental than essential

inheritance by aggregation. This proves to be the case. Suppose that class C has the parent classes D_1, ..., D_n, O is an instance of C, and P_i (i = 1, ..., n) are the corresponding instances of the parent classes, like in the previous section. Incidental inheritance means that we do not want to say that C *is-a* D_i, not even that C *like* D_i [WegZd]. Therefore, a client of O will not expect to get from it any of the services provided by D_i objects. We certainly intend that C *is-a* C^-, so we must not count $C^- = D_0$ as a parent class in this section!

I found a striking parallelism between [Snyd2,3] and [Lieb&al, LiHoRi, Sakk2] when parent classes in the former papers are equated with component classes in the latter ones. As regards *attributes* (instance variables), Snyder's requirement that instance variables of a parent class should not be directly accessible in a child class is equivalent with the rule in the Law of Demeter™ that a class's methods should see only the immediate structure (components) of the class. (Snyder's requirement as such is also presented in [LiHoRi] as an additional rule that makes the difference between the *strong* and the *weak* Law of Demeter.) As regards *operations* (instance methods), Snyder's requirement that "a class may refer to non-immediate ancestors only if they are exposed via the intervening classes" is again equivalent with a rule in the Law of Demeter: operations of the components of a class must not be directly invoked from outside the class.

In the implementation of CommonObjects [Snyd1], our complex object O would be a tree of subobjects such that O^- is its root and P_1, ..., P_n are its leaves. This is the most logical representation; we will just regard the subobjects as first-class objects too.

Neither multiple inheritance nor even repeated inheritance of the same parent [Meye2] presents many additional problems in this approach. In the terms of [LKnu], only casual horizontal name collisions can occur, and these can be resolved by qualification. Thus at worst, O must request any otherwise ambiguous inherited operation from an explicitly specified parent object. Both Snyder and Demeter require that the *attributes* of a parent component P_i not be directly available to (the operations of) O, and the *operations* of P_i not be directly available to other components nor clients of O. This does not prevent O from exporting any or all of P_i's operations to O's clients, renaming them or not, but the clients will see them as O's own operations.

Our approach actually comes close to *delegation* here (cf. [Snyd3] §2.3) in that we model the inherited parts as objects in their own right. We make the significant difference to conventional delegation that no request within a P_i for any of its own operations shall be dynamically changed into a request for an operation of O (O^-) with the same name, because we do not regard it as the "same" operation. This, I think, closes the last leak that still remained in [Snyd3] in encapsulating inheritance. Late binding of operations,

---

Demeter is a trademark of Northeastern University.

which many languages enforce in all situations, can make a parent class D_i in a way a client of the child class C: encapsulation is severely violated. Weird situations are possible even in a generally well-disciplined language like Eiffel ([Meye2] §11.2): If an operation *f* of class D_i is redefined in class C, the original *f* can be renamed and invoked within C by the new name. But there is no way whatsoever for other operations of D_i to invoke the original *f*, guaranteedly uninterfered by any later subclass definition.

Many languages, including CommonObjects, do offer a means to specify that some particular operation invocation is to be bound early (to the class of the *invoking* operation). Some languages, at least Simula and C++, additionally allow an ancestor class to declare which of its operations can be redefined in descendant classes (are **virtual**) and which not. BETA further allows a descendant class to modify an inherited virtual function to non-virtual for its own descendants, which is a very logical possibility. Here we have a situation different from all these: early binding of D_i's operations is a property of C, not of D_i itself, and is a consequence of the inheritance from D_i to C being strictly incidental and therefore encapsulated.

A property distinct from (but not quite independent of) the redefinability of operations is their *visibility*. One language that gives a good control of operation (and attribute) visibility is Trellis/Owl. Likewise, current C++ [Stro2] has the alternatives: **private**, visible only within the class itself; **protected**, visible to subclasses also; **public**, visible everywhere in the programme. In the special case of incidental inheritance, however, we suggest that even protected operations of D_i should *not* be invocable in C — since C is not a *subtype* of D_i. C++ originally had only the alternatives *private* and *public*. Some others, including Eiffel, have only *protected* and *public*, which I consider a worse selection. Unfortunately all operations are public in some OOPL's, including Smalltalk.

Incidental inheritance provokes the question about the visibility of *classes* [BKri&al, BuhZa]. The Law of Demeter also restricts the accessibility of classes to each other, but rather with "need to know" principles, not on the basis of nesting. If a class D_i is designed only to support C and perhaps some further class E, then the very existence of D_i could well be hidden from the clients of C and E.

7 MODELLING ESSENTIAL INHERITANCE

After the completely incidental inheritance in the previous section, we will now examine the other extreme. This is essential inheritance with the very strong requirement of *complete compatibility* [WegZd]:

> A subtype is completely compatible with its supertype if it has the same domain as the supertype and, for all operations of the supertype, corresponding arguments yield corresponding results.

("The same domain" here does not preclude the subtype from possessing additional attributes and operations besides the inherited ones.) As shown in [WegZd], this implies the

principle of substitutability:

> An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

Because we have wanted to encapsulate attributes, we could omit the "same domain" requirement from the first definition; this obviously would not disturb the principle of substitutability. If we were more interested in specifications than implementations, the definition could even better be expressed in terms of *traces* [McLe].

Let C be a subclass of D_i, as in the previous sections, but now with complete compatibility (CC). That requires every public operation *f* of D_i to be available also in C. Let again O be an instance of C and P_i the corresponding instance of D_i. If any call of O.*f* is just directed to the D_i component as a call to P_i.*f* with the same argument list, we trivially obtain CC. Are there any simple rules that could ensure CC in non-trivial situations? We propose the following sufficient but not necessary condition:

(1) The subclass operation O.*f* must invoke the superclass operation P_i.*f* and return the result given by P_i.*f*. Other actions of O.*f* must not modify P_i.

If there are *subclass-visible* (protected) operations, we need an additional rule (again sufficient but not necessary):

(2) No subclass operation must directly invoke any *modifying* protected operation of the superclass.

This is because such protected operations might leave the superclass object in a state that could not result from applying only public operations. It is also logical to require that all protected operations of the superclass remain protected operations of the subclass. We may allow them to be redefined subject to condition (1); condition (2) then implies that this is only possible with non-modifying protected operations.

If some subclass operations do not obey the rules (1) and (2), it might still be possible to prove complete compatibility e.g. along the lines of [Lamb], using traces. A method that employs traditional pre- and postconditions together with abstraction functions to prove behavioural compatibility is presented in [Amer2].

The complex object structure required by essential inheritance can be a little different from the pure tree structure suggested in the previous section. In an extreme case, the root object O^- and thus also the class C^- can be omitted entirely: if O is merely a "sum" of P_1, ..., P_n and no two distinct ancestor objects have any common public or protected operation. (The next section will present situations in which *one* ancestor object can be reached by different inheritance paths.) *This means that we can add "orthogonal" classes to existing classes in essential inheritance without adding levels to subobject trees, i.e. without*

introducing more levels of abstraction. Normally, the subclass C redefines and adds some operations, of course; a tree structure *is* then needed. However, all other operations of the parent classes remain as visible in the context of C as they originally were.

We have seen that complete compatibility requires the opposite on the visibility of inherited operations, when compared to incidental inheritance. (Obviously, we must not prohibit late binding of superclass operations, as we did in the previous section: all operations must be searched from the complex object level.) In our aggregation model we seem to get a conflict with the Law of Demeter by looking more than one level deep into the subobject tree. We can resolve the conflict by specifically allowing this exception to the law; but the need to make an exception should make us cautious about essential (visible) inheritance.

In typical cases of essential inheritance, the CC property cannot be required. This means that public operations of a descendant class may return different results than those of an ancestor class, and not all public operations of an ancestor are even offered by a descendant. This again implies that run-time checks may be necessary even in strongly typed languages. An important case is *read-only substitutability*, defined in [WegZd] as follows:

> An instance of a subtype can always be used in read-only mode in any context in which an instance of a supertype was expected.

The difference to CC is that updating operations of an ancestor may work incompatibly with a descendant (as long as they preserve any invariants that the ancestor class may have) or need not be applicable at all: the extreme case is a constant object. 'Read-only' can be interpreted as 'having benign side effects' [Lamb].

Repeated essential inheritance of a parent class is not straightforward. Although I do not know of any other language than Eiffel allowing direct repeated inheritance, it can easily arise indirectly in any language that permits multiple inheritance (cf. next section). If C inherits D *m* times, we cannot say that C *is-a* D; instead, C *is-a set* of D, although a set with a fixed number of elements. Handling such a situation in an orderly way would require intrinsic set operations, which most OOPL's have not got. Practically all languages impose some conditions, e.g. on explicit renaming or implicit operation lookup, that effectively leave at most one path of inheritance essential in our sense.

8 FORK-JOIN INHERITANCE

Explaining essential inheritance by aggregation gets tough when a class inherits a non-immediate ancestor over more than one path. For lack of a generally agreed term, we will call this situation *'fork-join inheritance'*. The most simple example possible is sufficient to show the difficulties: let D and E be parents of C, and F a parent of both D and E. We will look at how it is handled in Eiffel ([Meye2] §11.6.2), in the coming (?) version of C++ with multiple inheritance ([Stro2] §3), and in the theoretical paper [LKnu]. The last paper does not discuss operations, which are for our purposes both more important and more difficult than attributes.

Note that we get no new kind of problem if one or more of the four inheritance links involved is totally incidental: C cannot then "see" F over two paths. Otherwise, if the situation we want to model is such that the D and E subobjects of C should each have its own private F subobject, we have repeated inheritance (see end of previous section), just indirectly. The new, interesting case arises when we require a shared F subobject, thus a fork-join relationship on the instance level, too.

Stroustrup's model allows every class to declare any or all of its parents **virtual**. If F is declared virtual in both D and E, it means that the D and E parts of a C object share a common F part; otherwise each has its own. (In the example of [Stro2], C declares F also as a (direct) **virtual** parent, but this can hardly be necessary in general.) The approach is clean and understandable; most importantly, it preserves the integrity of the F subobject. It fits the model of §5 perfectly. Nevertheless, as discussed in [Stro2], programming the operations in fork-join inheritance is somewhat tricky: each class will typically need, for each public operation, an accompanying *protected* operation (with a different name) that does only "the own stuff" of the particular class. The public operations of each class must then explicitly call the appropriate protected operations of all ancestor classes. — Built-in modes of operation combination in Flavors [Moon] and CLOS [Bobr&al] probably eliminate such programming chores in most situations.

Meyer presents a "transcontinental drivers" example that cannot be simply and naturally reduced into object aggregation. Part of the attributes inherited from F by D and E are shared, part replicated. Those that are replicated are renamed in C so that there are no name conflicts: the repeated inheritance rule of Eiffel says that exactly those inherited attributes and operations shall be shared that have *not* been renamed along any of the inheritance paths. The problem is that the F part gets effectively split into two. The integrity of subobjects is thus violated, somewhat like in Flavors (§5).

Lindskov Knudsen suggests the responsibility for sharing or replicating to be divided between C and F: just the opposite of the C++ approach. Every (non-inherited) attribute of every class must be declared either *singular* or *plural*; singular attributes will always be shared in a fork-join inheritance. Every class must declare whether its inheritance method is *unification* or *intersection*. If it is unification, then any plural attribute inherited along several paths from a common ancestor is replicated. If it is intersection, then even such plural attributes are shared. Clearly, the integrity of subobjects can get violated in this approach, too, if we try to reduce inheritance to aggregation.

I claim that these latter two inheritance models allow anomalies that make them not generally recommendable. Obviously, the approach of [Meye2] is strictly more general (or less disciplined) than that of [LKnu]. The same kind of anomaly still creeps up in both models. The really fatal defect is that any operation of F, D, or E, public, protected, or private, *that both updates shared attributes and accesses replicated attributes, may cause unwanted side effects between the D and E parts of a C object. All operations of F, D, and E must therefore be checked, and the code-sharing advantage of inheritance is lost.*

Another argument is that the examples given to illustrate these two inheritance models look like patches to bad class design in the ancestor class. A relational database expert would probably identify the heart of the problem as F not being in second normal form. We look at the example of [Meye2], which is smaller. The "root" class (F) is called Driver: it contains attributes such as Age and Number-of-violations. The intermediate classes are called French-driver and US-driver: both inherit all attributes of Driver and have some of their own. The lowermost class is called French-US-driver: it inherits all attributes of both French-driver and US-driver. Some Driver attributes are shared, e.g. Age, some are renamed and thus replicated, e.g. Number-of-French-violations, Number-of-US-violations.

We note that French-US-driver is not a *subtype* of any of its ancestor classes, because it has not all their attributes. It would be a subtype of two of them if all replicated Driver attributes were renamed only on one path; but then French-driver and US-driver would not be in an equal position. The whole class seems to have very little purpose other than to coerce the splitting of Driver. The approach begins to look futile indeed when we consider that every combination of two or more countries that is needed to model some multinational driver in the system requires a similar class of its own.

I propose that the Driver class should be explicitly divided into two classes in the first place: Person (containing Age and the other attributes that are now shared) and Only-driver (= Driver – Person, containing Number-of-violations and the other replicated attributes), such that Person is a shared (virtual) parent of Only-driver. French-driver and US-driver would inherit (without sharing) from Only-driver. In fact, it could be better not to regard the relationship of Person and Only-driver as inheritance at all, but merely to declare an attribute of type '[reference to] Person' in Only-driver. That way, we need not necessarily delete a person from the system just because he loses his driving licence. — In the model of [LKnu], every class that has both singular and plural attributes should similarly be split into two classes. Both unification and intersection can then be handled within our model.

9 TOPICS TO BE PURSUED

There are some aspects connected to the theme of this contribution that I already have ideas about or that should be taken into account before the model of inheritance as aggregation is completed. Some thoughts surfaced too late to be developed, when finishing this paper for publication. Besides, the paper is too long already, so a short listing shall suffice.

In multiple inheritance, some ancestors of a class may be incidental and some others essential. The discussion of sections 6 and 7 is easily applicable to this case. A problem that we have not treated is a partly incidental, partly essential inheritance relationship between a subclass and *one* parent class: for instance, only part of the parent's public operations are *public in the child as well. We have not spoken much even about essential inheritance weaker than completely compatible. Errors and misunderstandings in programming may most easily happen in this "gray area". I feel that it could perhaps be possible to*

"normalise" classes into difference classes (very analogously to database normalisation) such that inheritance could be factored into purely incidental relationships and CC essential relationships. (Cf. "consistent subsets" and "inheritance packages" in [Amer1] §3.3.)

In contrast to the above, the boundary between incidental inheritance and ordinary aggregation does not look problematic at all. Any or all of the restrictions (1-3) in §5 can be relaxed or removed without causing obvious inconsistency. For instance, the composite objects of ORION [Kim&al] need not obey rules (2) and (3) nor the "only when" part of rule (1).

In my opinion, any self-respecting framework of multiple inheritance must offer an elegant and consistent treatment of "multi-methods" [Bobr&al, Øste]. This can be difficult, e.g. because the "specificity" order of class combinations is only partial.

It seems that the current approach can solve "the *self* problem" as presented in [Lieb1] §6. I think it can be worthwhile for some purposes to introduce at least two further reflexive pseudo-variables ("reflexive pronouns"), both usually denoting a smaller complex object than *self*, but mostly larger than *super*. The first one would denote the subobject to which the current operation belongs (*my-node* in CommonObjects [Snyd1]). The other one would denote the subobject whose operation has been originally called from outside the complex object; it would be interesting only if restriction (3) of §5 is omitted.

The importance of *names* (identifiers invented by the programmer) in identifying operations (also attributes and classes) can and should be diminished. We hinted at this in §4; many name clashes are simply false friends that can be thus avoided. The use of *titles* (roles) instead looks promising; titles can also be employed to create objects that are an intermediate of class and instance [Sakk4].

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (initially) and the Ministry of Education (doctoral programme in information technology). Correspondence with the Demeter group of Northeastern University (Karl Lieberherr, Ian Holland, and others) has partially inspired me to this particular line of research.

I am much indebted to Ian Holland for a thorough commentary on the first version of this paper; it has caused many revisions. The conference reviewers' comments were useful, too. I also thank Alan Snyder (Hewlett-Packard Laboratories) for clarifying some points, and Pierre America (Philips Laboratories) for providing some relevant papers.

REFERENCES

[AghHe] Gul Agha and Carl Hewitt, *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*, [ShrWe] 49-74.

[Amer1] Pierre America, *Inheritance and subtyping in a parallel object-oriented language*, [Bézi&al] 234-242.

[Amer2] Pierre America, *A Behavioural Approach to Subtyping in Object-Oriented Programming Languages*, Philips Research Laboratories, Eindhoven (The Netherlands) 1989.

[AnPoWa] M. S. Anderson, R. D. Pose, and C. S. Wallace, *A Password Capability System*, Comp. J. 29:1 (1986) 1-8.

[Bézi&al] Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman (Ed.), *ECOOP '87 European Conference on Object Oriented Programming (Paris, June 1987) Proceedings*, Lecture Notes in Computer Science 276, Springer-Verlag 1987.

[Blac&al] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy, *Object Structure in the Emerald System*, [Meyr1] 78-86.

[Bobr&al] Daniel G. Bobrow et al., *Common Lisp Object System Specification*, ACM SIGPLAN Notices 23: special issue (September 1988).

[Born] Alan Borning, *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM ToPLaS 3:4 (October 1981) 353-387.

[BKri&al] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard, *The BETA Programming Language*, [ShrWe] 7-48.

[BuhZa] P. A. Buhr and C. R. Zarnke, *Nesting in an Object-Oriented Language is NOT for the Birds*, [GjeNy] 128-145.

[CaDeWa] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg, *A Data Model and Query Language for EXODUS*, [SI88] 413-423.

[ChiD'A] E. Chiricozzi and A. D'Amico (Ed.), *International Conference on Parallel Processing and Applications (L'Aquila, Italy, September 1987) Proceedings*, North-Holland 1988.

[CorLe] Antonio Corradi and Letizia Leonardi, *How to embed concurrency within an object environment: Parallel Objects*, [ChiD'A] 79-84.

[DaMyNy] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard, *SIMULA 67 Common Base Language*, Norwegian Computing Center 1968 (No. S-2).

[GjeNy] S. Gjessing and K. Nygaard (Ed.), *ECOOP '88 European Conference on Object Oriented Programming (Oslo, August 1988) Proceedings*, Lecture Notes in Computer Science 322, Springer-Verlag 1988.

[GolRo] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.

[HaiNg] Brent Hailpern and Van Nguyen, *A Model for Object-Based Inheritance*, [ShrWe] 147-164.

[JohFo] Ralph E. Johnson and Brian Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming 1:2 (June/July 1988) 22-30,35.

[Kim&al] Won Kim et al., *Composite Object Support in an Object-Oriented Database System*, [Meyr2] 118-125.

[LaThPu] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh, *An Exemplar Based Smalltalk*, [Meyr1] 322-330.

[Lamb] David Alex Lamb, *Benign Side Effects*, Inf. Proc. Lett. 29:6 (December 1988) 301-305.

[Lieb&al] Karl Lieberherr, Ian Holland, Gar-lin Lee, and Arthur J. Riel, *An objective sense of style*, Computer (IEEE) 21:6 (June 1988) 79-81 (The Open Channel).

[LiHoRi] K. Lieberherr, I. Holland, and A. Riel, *Object-Oriented Programming: An Objective Sense of Style*, [Meyr3] 323-334.

[LieHo] Karl J. Lieberherr and Ian Holland, *Formulations and Benefits of the Law of Demeter*, submitted paper (1988).

[Lieb1] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, [Meyr1] 214-223.

[Lieb2] Henry Lieberman, *Concurrent Object-Oriented Programming in Act 1*, [YonTo] 9-36.

[LKnu] Jørgen Lindskov Knudsen, *Name Collision in Multiple Classification Hierarchies*, [GjeNy] 93-109.

[Lisk&al] Barbara Liskov et al., *CLU Reference Manual*, Lecture Notes in Computer Science 114, Springer-Verlag 1981.

[McLe] John McLean, *A Formal Method for the Abstract Specification of Software*, JACM 31:3 (July 1984) 600-627.

[Meye1] Bertrand Meyer, *Genericity versus Inheritance*, [Meyr1] 391-405.

[Meye2] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall 1988.

[Meyr1] Norman Meyrowitz (Ed.), *OOPSLA '86 Conference Proceedings (Portland, Oregon, 1986)*, ACM SIGPLAN Notices 21:11 (November 1986).

[Meyr2] Norman Meyrowitz (Ed.), *OOPSLA '87 Conference Proceedings (Orlando, Florida, 1987)*, ACM SIGPLAN Notices 22:12 (December 1987).

[Meyr3] Norman Meyrowitz (Ed.), *OOPSLA '88 Conference Proceedings (San Diego, California, 1988)*, ACM SIGPLAN Notices 23:11 (November 1988).

[MinRo] Naftaly H. Minsky and David Rozenshtein, *A Law-Based Approach to Object-Oriented Programming*, [Meyr2] 482-493.

[Moon] David A. Moon, *Object-Oriented Programming with Flavors*, [Meyr1] 1-8.

[Øste] Kasper Østerbye, *Abstract Data Types with Shared Operations*, ACM SIGPLAN Notices 23:6 (June 1988) 91-96.

[Sakk1] Markku Sakkinen, *On the darker side of C++*, [GjeNy] 162-176.

[Sakk2] Markku Sakkinen, *Comments on "the Law of Demeter" and C++*, ACM SIGPLAN Notices 23:12 (December 1988) 38-44.

[Sakk3] Markku Sakkinen, *Objects, non-objects, and existential dependences*, submitted paper (1988).

[Sakk4] Markku Sakkinen, *Between classes and instances, aided by titles*, submitted paper (1989).

[Scha&al] Craig Schaffert et al., *An Introduction to Trellis/Owl*, [Meyr1] 9-16.

[ShrWe] Bruce Shriver and Peter Wegner (Ed.), *Research Directions in Object-Oriented Programming*, The MIT Press 1987.

[SI88] *SIGMOD '88 Conference (Chicago, June 1988) Proceedings*, ACM SIGMOD Record 17:3 (September 1988).

[SkaZd] Andrea H. Skarra and Stanley B. Zdonik, *Type Evolution in an Object-Oriented Database*, [ShrWe] 393-415.

[Snyd1] Alan Snyder, *CommonObjects: An Overview*, ACM SIGPLAN Notices 21:10 (October 1986) 19-28.

[Snyd2] Alan Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, [Meyr1] 38-45.

[Snyd3] Alan Snyder, *Inheritance and the Development of Encapsulated Software Systems*, [ShrWe] 165-188.

[Stei1] Lynn Andrea Stein, *Delegation Is Inheritance*, [Meyr2] 138-146.

[Stei2] Lynn Andrea Stein, *Compound Type Expressions: Flexible Types in Object Oriented Programming*, [Meyr3] 360-361.

[Stro1] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.

[Stro2] Bjarne Stroustrup, *The Evolution of C++ : 1985 to 1987*, USENIX C++ Workshop (Santa Fe, New Mexico, November 1987) Proceedings.

[UngSm] David Ungar and Randall B. Smith, *Self: The Power of Simplicity*, [Meyr2] 227-242.

[Viha] Juha Vihavainen, *The Programming Language Mode Language Definition and User Guide*, Report C-1987-50, University of Helsinki (Finland), Department of Computer Science 1987.

[WegZd] Peter Wegner and Stanley B. Zdonik, *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*, [GjeNy] 55-77.

[YokTo] Yasuhiko Yokote and Mario Tokoro, *Concurrent Programming in ConcurrentSmalltalk*, [YonTo] 129-158.

[YonTo] Akinori Yonezawa and Mario Tokoro (Ed.), *Object-Oriented Concurrent Programming*, The MIT Press 1987.