Title:

Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems.

Authors:

Marc Van Limberghen
Department of Computer Science
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: mvlimber@vnet3.vub.ac.be
Phone: (32) 2 6293492
Fax: (32) 2 6293495
(responsible for correspondence)

Tom Mens
Department of Mathematics
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: tommens@is1.vub.ac.be
Phone: (32) 2 6293474
Fax: (32) 2 6293495

Abstract:

In class-based multiple inheritance systems, interpretations as different as duplication, sharing and specialisation are associated with the single phenomenon of name collisions. To deal with those name collisions, various mechanisms have been proposed. But these solutions generally restrain software reusability which is considered to be one of the key features of OO systems. On top of this, most multiple inheritance systems do not completely cover all the different interpretations of name collisions.

This paper shows that the entire multiple inheritance dilemma can and should be seen as a conflict between inheritance and data encapsulation only. Orthogonalising these two concepts in a mixin-based framework will permit us to appropriately solve all the problems listed above. To this extent we propose a formal model together with its denotational semantics. This minimal multiple inheritance model establishes a valuable basis for OO languages and software engineering systems.

Keywords:

Multiple Inheritance, Name Collisions, Mixins, Encapsulation, Composition.

Running title:

Solving multiple inheritance name collisions using mixins.

Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems.

In class-based multiple inheritance systems, interpretations as different as duplication, sharing and specialisation are associated with the single phenomenon of name collisions. To deal with those name collisions, various mechanisms have been proposed. But these solutions generally restrain software reusability which is considered to be one of the key features of OO systems. On top of this, most multiple inheritance systems do not completely cover all the different interpretations of name collisions.

This paper shows that the entire multiple inheritance dilemma can and should be seen as a conflict between inheritance and data encapsulation only. Orthogonalising these two concepts in a mixin-based framework will permit us to appropriately solve all the problems listed above. To this extent we propose a formal model together with its denotational semantics. This minimal multiple inheritance model establishes a valuable basis for OO languages and software engineering systems.

Keywords: Multiple Inheritance, Name Collisions, Mixins, Encapsulation, Composition.

1. Introduction

In class-based languages that use multiple inheritance, multiple parents of a class can have instance variables or methods with the same name. The question arises how to treat these name collisions. Different semantics to deal with those collisions are needed: specialisation, sharing and duplication. The diversity of these semantics causes a lot of problems for current multiple inheritance systems.

Knudsen (1988) fully described the problems concerning duplication of instance variables of shared ancestors. We will generalise these problems by taking also methods into account. Carré & Geib (1990) investigated another problem, namely the preservation of homonymous attributes of multiple, independent superclasses. They rejected message selector renaming and class qualification as solution mechanisms. Snyder (1987) indicated problems resulting from the exposure of inheritance structure. Sakkinen (1989) discussed the problem of unwanted side effects between independently developed subclasses of a single parent.

Historically these different kinds of problems were investigated one by one as they emerged. Research literature mostly concentrated on only one of them, and suggested a solution specific to one category while ignoring the others. Some systems simply gathered different solution mechanisms, but -as far as we know- never succeeded in accurately solving all the problems.

We will involve another category of problems in the field of multiple inheritance, namely encapsulation (in the sense of attribute visibility). We will show that the mentioned problems are closely related. We will exploit this similarity to uniformly solve all problems with a single strategy, namely an orthogonal and disciplined combination of mixin-based inheritance and encapsulation:

- Mixin-based inheritance (Bracha & Cook, 1990) can accurately solve duplication problems. It also deals with the exposure of inheritance structure in the sense that it gives rise to a layered software engineering scheme.
- Encapsulation of methods is often only considered with regard to message passing clients. However an encapsulation mechanism can also be very useful for inheritors: introducing scope in the inheritance hierarchy avoids name collisions solving the problems concerning homonymous characteristics. Moreover encapsulation towards inheritors favours the independent development of subclasses.

The key to solve the different multiple inheritance problems lies in the combination of both concepts.

Bracha & Lindstrom (1992) showed that viewing inheritance as a composition of software modules can be regarded as a new way of building adaptable software. We will introduce an encapsulation mechanism similar to their *hide* operator. Together with the usual inheritance operator, this encapsulation operator is appropriate and sufficient to deal with multiple inheritance problems. No additional operators are required.

In the evolution of object-oriented programming two groups of languages have emerged: class-based languages and prototype-based (or object-based) languages. Class-based languages define inheritance on classes while prototype-based languages do this on objects. The problems mentioned will be described in a class-based medium, but are also present in prototype-based languages that try to support multiple inheritance.

The class-prototype controversy however concerns a much larger dilemma than the absence or presence of classes. Prototype-based languages are generally considered too flexible resulting in a mere code-sharing mechanism without conceptual meaning. The class-based mixin model we will present suffers from the same criticism since mixins are code components that can be freely composed. Our model should however be seen as a basis that can be orthogonally extended with suitable software engineering tools reinforcing reliability, as will be clarified in section 7.

2. Multiple inheritance problems

In this section we will illustrate the common multiple inheritance problems with program examples. To this extent we first need to agree upon some terminology and notational conventions. In the rest of this text the term *attribute* will designate an instance variable or a method. We will also adopt the following message passing notation in our examples:

- Messages with an explicit receiver are denoted by writing the receiver followed by the message-selector, that we will call *label*. Arguments can be passed by putting them behind the label between parentheses. E.g. *p move(1,2)* means sending the message *move* to *p* with arguments *1* and *2*.
- Receiverless messages represent self sends, e.g. *x* means *self x*
- To avoid confusion with receiverless self sends, formal argument names are always preceded by a # symbol.
- Super calls are denoted with the *super* keyword. We use super calls only to refer to the method we are currently overriding. Consequently it is redundant to specify which super method is activated. In other words when for example overriding the *move* method, we will no longer write *super move(1,2)* to call the super variant of the *move* method, but simply *super(1,2)*.
- We have included instance variables in a slot-based way (Snyder, 1987): instance variables can only be accessed through a pair of *accessor methods,* even from within the class: a retrieve method that returns the value of the instance variable, and an update method that expects an argument and updates the value of the variable with this argument. From the point of view of the client (inheritor as well as message passing client), accessor methods must behave in exactly the same way as ordinary methods: a client should not be aware of whether she is invoking state or behaviour. This obviously leads to a higher degree of abstraction and modifiability, making inheritance more powerful. To make the examples more clear we will adopt the convention that, given a certain instance variable *x*, its accessor methods both have the same label *x*. Nevertheless they can be distinguished from each other since the update method expects an argument between parentheses, while the retrieve method does not.

2.1. Duplication of common ancestors

A first multiple inheritance problem arises when different parents of a class have common ancestors. Should the instance variables in those common ancestors be duplicated or shared? Duplication means simultaneously holding multiple versions of the instance variable. A similar question holds for a method of the common ancestor. Should the associated method-body be invoked once or several times when the method is selected? The following example illustrates a class in which a combination of both duplication and sharing (respectively multiple and single invocation) is desired.

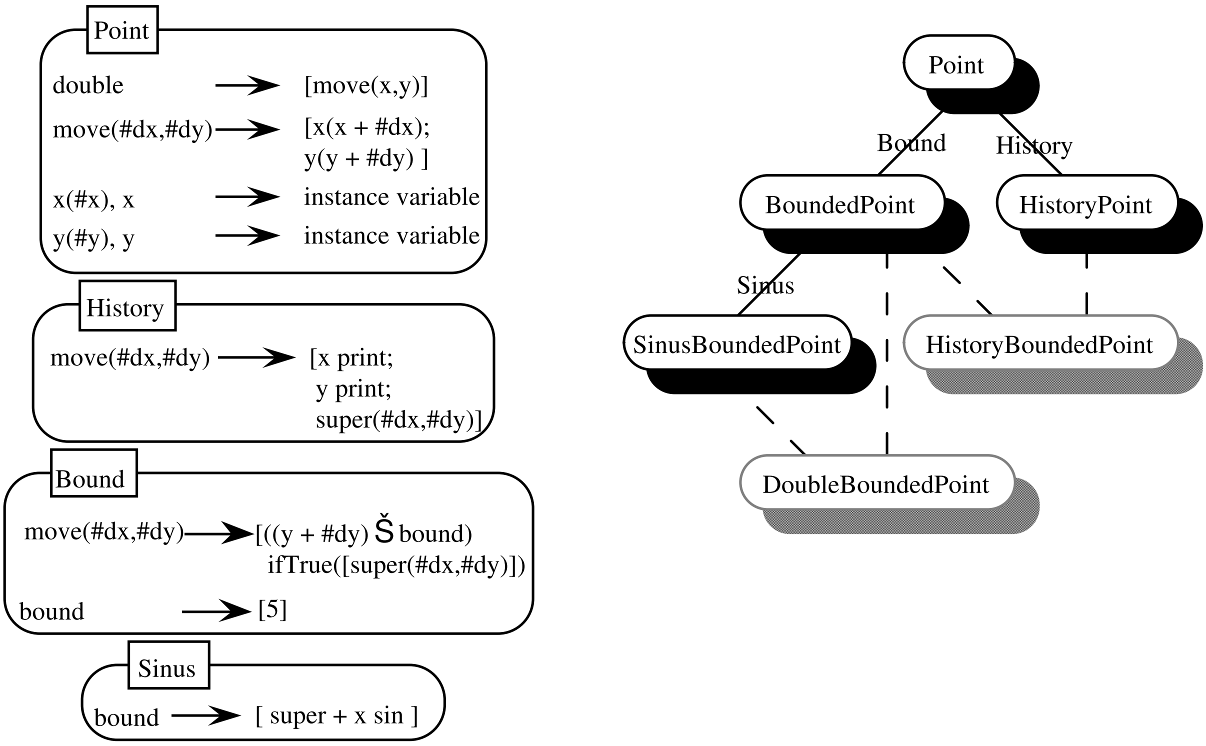

Figure 1: Point class hierarchy and corresponding code

Consider the example depicted by the class hierarchy in Figure 1. The black shadowed rounded rectangles can be seen as traditional classes. The code of the root class **Point** and its subclass extensions are presented on the left. A **Point** is a class containing two instance variables *x* and *y*, a *move* method that moves the point, and a *double* method that doubles the *x* and *y*-values of the point by making use of the *move* method. Suppose we want to obtain a **BoundedPoint** class, that ensures that the *y*-value of the point never exceeds a certain constant upper bound. Then we need to override the existing *move* method with one that performs an additional check. The upper bound does not even have to be a constant value, but can be an arbitrary function, e.g. a sinus. To illustrate this, we will extend the **BoundedPoint** class to a **SinusBoundedPoint** class by overriding the *bound* method in an appropriate way. Invoking the *move* method on an instance of this **SinusBoundedPoint** class moves the point, and checks if the new ordinate of the point lies under the curve illustrated in Figure 2.

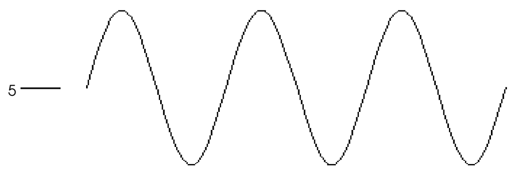

Figure 2: An upper sinus bound

Another specialisation of **Point** is **HistoryPoint**. Its instances print the old value of *x* and *y* every time the *move* method is invoked.

Until now we haven't encountered any problems, since we didn't multiply inherit yet. The grey shadowed rounded rectangles in Figure 1 denote classes we would like to construct with some multiple inheritance mechanism. Suppose that we want to create a **HistoryBoundedPoint** class that includes the functionalities of both **BoundedPoint** and **HistoryPoint**. It is clear that the *x* and *y* instance variables and the *move* method that occurred in the **Point** class should be shared in the new **HistoryBoundedPoint** class. The two different *move* specialisations have to be combined somehow.

Consider on the other hand a **DoubleBoundedPoint** class, that ensures that the upper bound of a certain **Point** never exceeds two upper bounds: a sinus bound and a horizontal bound. In other words, we want to check if the *y*-value of a point always lies under the curve depicted in Figure 3.

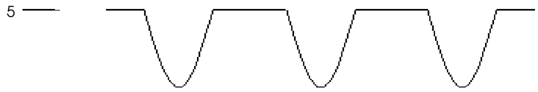

Figure 3: Composition of two upper bounds

DoubleBoundedPoint needs two versions of the *bound* attribute, once as a constant, once as a sinus. Consequently the *bound* attribute must be duplicated. The *move* method-body of Bound needs to be invoked twice: once with each *bound* version. Hence **DoubleBoundedPoint** needs duplication of the attributes introduced by the common ancestor **BoundedPoint**, while the attributes of the common ancestor **Point** need to be shared. (Knudsen, 1988) even gives an example where an instance variable of the same ancestor common to n parent classes needs to be duplicated less than n times. The programmer thus needs a flexible control concerning sharing and duplication.

2.2. Homonymous attributes

The previous problem concerned multiply inheriting the same attribute. Another problem arises when we want to inherit different attributes with the same name, necessarily from different parents. One needs a mechanism to invoke each of these attributes separately.

Consider for example the **Person** class hierarchy in Figure 4. Some persons are entitled to reduction, but only if they are younger than 25 years. In this case they receive a reduction card, and belong to the **PersonWithReduction** class. A **Sportsman** also possesses a card, granting him access to the sport infrastructure. **TopSportsman** represents a sportsman with a special kind of sport card number. A sportsman that is entitled to reduction is represented by **SportsmanWithReduction**. She possesses two cards, one as sportsman and one to benefit from a reduction.

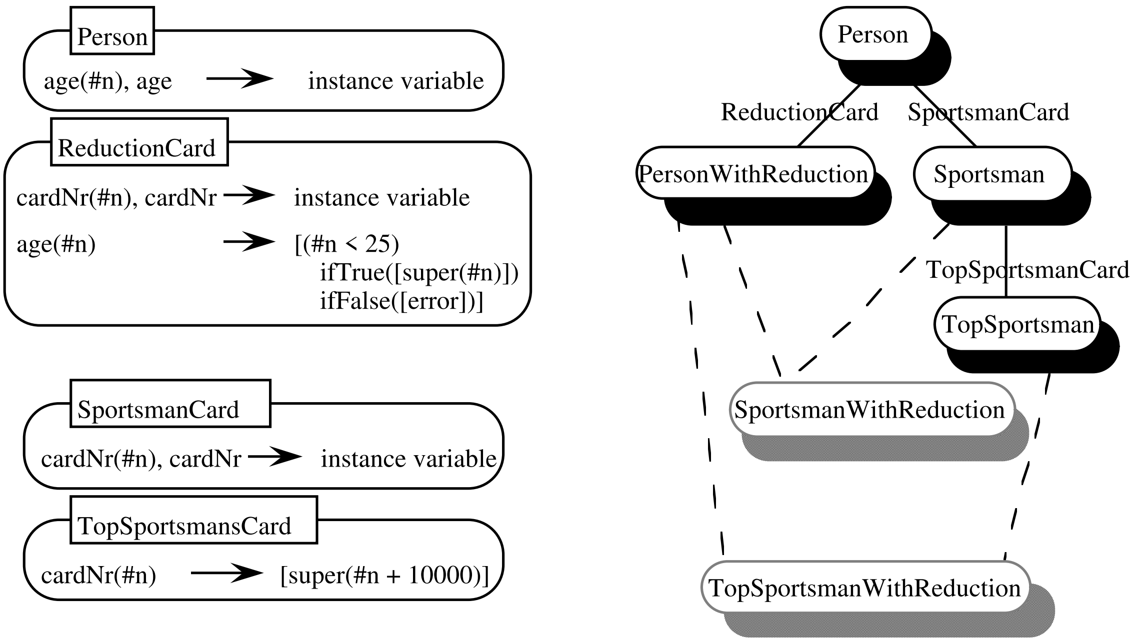

Figure 4: Person class hierarchy

A name collision between those different cards can appear by coincidence if both specialisations of **Person** have been designed independently. It can also be intentional, namely when a **Sportsman** and a **PersonWithReduction** must be mutually substitutable in some applications. For example in a program that fills in the card number of a person via a dialog box that can be used for both kinds of card number.

Suppose that we want the class **SportsmanWithReduction** to provide a method *setCardNumbers* that updates both card numbers. In this case we need to access both card numbers separately. A multiple inheritance system should offer this access possibility. Moreover such an access strategy should take the so-called genericity inhibition problem of Carré & Geib (1990) into account. In this case this means that the setCardNumbers code of **SportsmanWithReduction** should also be reusable for **TopSportsmanWithReduction.**

2.3. Encapsulation of attributes

In the literature the term encapsulation has been employed with different meanings. When we speak of encapsulation we always mean attribute hiding. Encapsulation of attributes is generally accepted as an important aspect of OO systems. Due to a conflict of interests the interaction between encapsulation and inheritance is very delicate. On one hand a subclass should have the possibility to override the implementation attributes of his superclass. On the other hand a subclass shouldn't have access to the implementation details of its parent, in order to guarantee the possibility to re-implement the superclass.

Most OO systems include encapsulation by distinguishing different kinds of attributes. In C++ (Ellis & Stroustrup, 1991) for instance *public* attributes are visible to all clients; *private* attributes are only locally visible (i.e. in the extension itself); *protected* attributes are invisible to message passing clients, but visible to inheriting clients (i.e. subclasses).

In the example of Figure 5, we define a **Lamp** as a class containing two public methods *dim* and *brighten* to regulate the *intensity* of the lamp. A **SafeLamp** is a special kind of lamp with a restricted intensity. A **ColouredLamp** is another kind of lamp of which the *colour* is given by a public instance variable. The *intensity* of the colour can only be adjusted via the public methods *darken* and *lighten*.

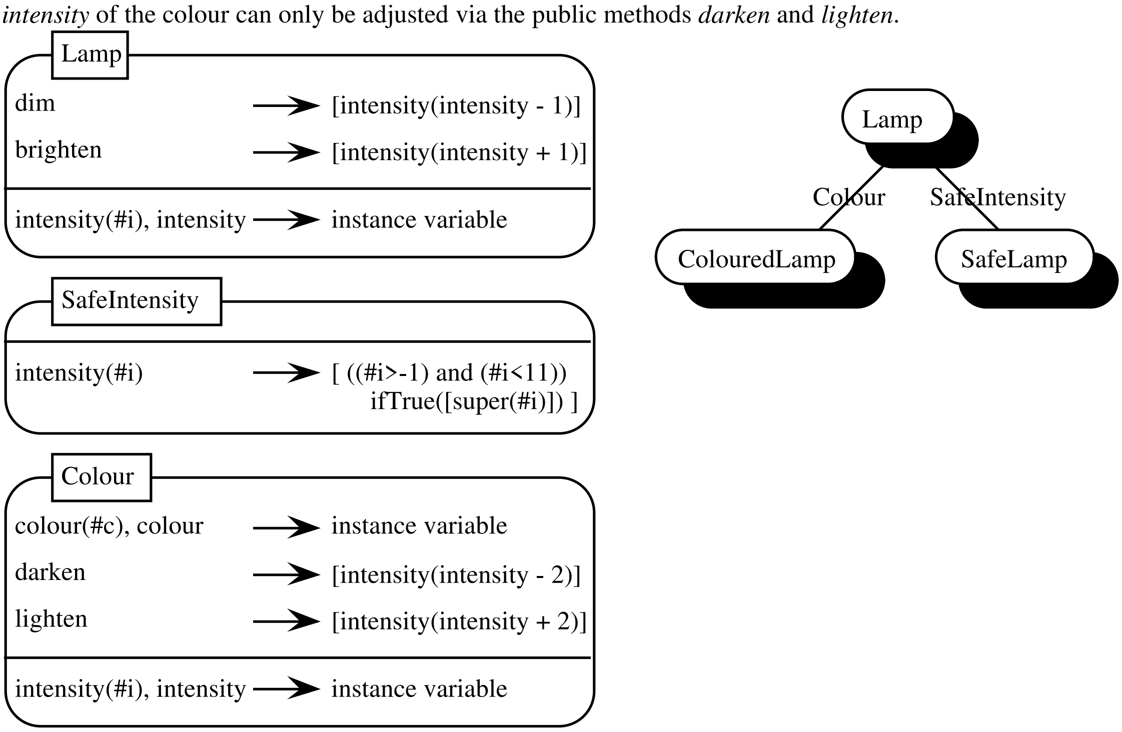

Figure 5: Lamp class hierarchy

In software engineering it is opportune to have the possibility to write the subclass and superclass code independently (after having agreed on the super calling and self sending protocol if necessary). This is also important in multiple inheritance to obtain a higher degree of substitutability of the superclass.

Due to independent development, Lamp and Colour may contain a homonymous implementation attribute *intensity*, denoting the intensity of the light and the intensity of the colour respectively. Since *intensity* is an implementation attribute (denoting an internal characteristic of the lamp), it shouldn't be visible for message passing clients. This in indicated in Figure 5 by a horizontal line separating the intensity from the other attributes. The question here is which kind of encapsulation is appropriate. Since the subclass **SafeLamp** overrides *intensity*, *intensity* should be declared as protected attribute in **Lamp**. The subclass **ColouredLamp** on the other hand coincidentally contains also an *intensity* attribute, requiring *intensity* to be declared as private attribute in **Lamp**. Consequently, in the context of independent development of software components, the use of protected and private attributes (as in C++) exhibits a conflict between data encapsulation and reusability interests. We need a more general encapsulation mechanism that allows the possibility to restrict visibility towards inheritors.

3. Current multiple inheritance systems

Current inheritance systems do not offer the required expressiveness to deal with the problems indicated in the previous section. We will now present a list of strategies commonly found in the literature, and show their shortcomings.

<u>Linear approaches</u> towards multiple inheritance automatically linearise the inheritance graph in order to handle name collisions. The linearisation algorithm never duplicates common ancestors and consequently fails to deal with the duplication problem. Flavors (Moon, 1986) and its successor CLOS (Keene, 1989) use linear approaches. Besides their impossibility to duplicate common ancestors, the two other problems discussed in section 2 are not treated.

<u>Tree multiple inheritance</u> on the other hand converts the inheritance graph in a tree by duplicating nodes that can be reached by multiple paths: in this case sharing of common ancestor attributes becomes impossible. Tree multiple inheritance is explored in CommonObjects (Snyder, 1987).

The <u>graph-based approach</u> was specifically meant to solve the homonymous attribute problem. By extending message passing with a class qualifier, this approach directly deals with the inheritance graph without transforming it. The method *setCardNumbers* of the class **SportsmanWithReduction** as mentioned in section 2.2 can then be implemented as follows:

```
setCardNumbers(#reductionNr, #sportsmanNr) →
        [PersonWithReduction.cardNr(#reductionNr); Sportsman.cardNr(#sportsmanNr)]
```

Due to this <u>class qualification</u>, method lookup starts from the specified class rather than from the class of the receiver of the message *setCardNumbers*.

Extended Smalltalk (Borning & Ingalls, 1982) extends Smalltalk with class qualification. However it ignores encapsulation and never duplicates common ancestors.

C++ also offers class qualification as an optional means to deal with homonymous attributes. Concerning the duplication problem, Knudsen (1988) indicates that the use of virtual classes in C++ is insufficient. As mentioned in section 2.3, C++ does not satisfy our software engineering requirements concerning encapsulation.

Carré & Geib (1990) criticised class qualification because it violates late binding and consequently restrains reusability. Reconsider for example the class **TopSportsmanWithReduction**. In order to implement this class, we need to rewrite the *setCardNumbers* method of **SportsmanWithReduction** because it refers to the wrong class! ROME, as presented by Carré & Geib, introduces as-expressions, an innovative solution to the homonymous attribute problem, but ignores duplication of common ancestors and encapsulation.

Message selector renaming is an alternative solution to the homonymous attribute problem. Rename clauses allow to rename inherited attributes. This way we can choose in the implementation of **SportsmanWithReduction** a different name for both inherited card numbers.

Eiffel (Meyer, 1988) belongs to the category of models using graph multiple inheritance. It offers message selector renaming to deal with homonymous attributes. Renaming is also used to deal with sharing and duplication. The renaming mechanism exhibits difficulties in constructing **TopSportsmanWithReduction** since the renaming clause explicitly refers to the inherited class name. This way class qualification problems turn up. Eiffel does not fulfil our requirements for encapsulation either since encapsulation towards inheriting clients is absent.

4. Disciplined encapsulated mixin-based inheritance

Research literature on multiple inheritance mostly concentrated on only one of the problems of section 2. Nevertheless these problems are apparently closely related. The *cardNr* in the person example and the *intensity* in the lamp example had to be duplicated, insinuating a similarity with the common ancestor duplication problem. On the other hand the *bound* attribute in the point example would be typically invisible to message passing clients, suggesting a relation to the encapsulation problem. Also the label *cardNr* in the person example may not be directly visible to instances of **SportsmanWithReduction** since there are two values according to that label.

This similarity not only appears between the problems, but also between respective candidate solutions. Section 2.3 showed that adding properties to attributes (public, private, protected) does not offer an adequate encapsulation mechanism. Also with regard to duplication adding properties to attributes is insufficient. This was illustrated by Knudsen (1988) who distinguished so-called singleton and plural instance variables. All these experiences indicate that distinguishing different kinds of attributes is an inferior strategy with respect to both duplication and encapsulation.

The novelty of our multiple inheritance approach is that we exploit this similarity to untangle the problems. Our model consists of disciplined mixin-based inheritance provided with an explicit encapsulation mechanism. On one hand, mixin-based inheritance fulfils the need of a more flexible control of sharing and duplication (pointed out in the common ancestor problem) by offering total expressiveness on the inheritance structure. On the other hand, an explicit encapsulation mechanism allows to limit the scope of attributes in the inheritance hierarchy. This way name collisions can be avoided.

4.1. History of mixin-based inheritance

Before starting to formalise our model, we will present an historic overview of mixins. Mixins first emerged in LISP-based languages like Flavors and CLOS. The concept of mixins is introduced there as a special use of the already present multiple inheritance mechanism. In this approach, a mixin is regarded as an abstract subclass, i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. More specifically a mixin is a class without specified superclasses, and is usually intended to support some aspect of behaviour orthogonal to the behaviour supported by other classes in the program (Keene, 1989).

In (Bracha & Cook, 1990) on the contrary, mixins are not a special use of, but rather the supporting mechanism for inheritance. For this reason the term *mixin-based inheritance* is coined. The essence of mixin-based inheritance is exactly to view mixins as stand-alone entities that can be used (or mixed in) in the construction of different classes. In this way each class is obtained as a composition of different mixins. The code components in the examples of section 2 can be seen as mixins, for example:

SinusBoundedPoint = Point *composed with* Bound *composed with* Sinus

where Point, Bound and Sinus are mixins corresponding to the code blocks in Figure 1.

Cook & Palsberg (1989) proposed a denotational semantics of the latter kind of mixins using the notion of wrappers, i.e. attribute records with an unbound self- and super-reference. Hense (1992) independently introduced a language complete with denotational semantics, containing wrappers as an explicit language feature.

An important difference between mixin-based inheritance and CLOS-mixins is the way parent classes are merged during inheritance.

- In CLOS the ordering of this merging is determined by a linearisation algorithm.[1] Each ancestor of a given class occurs only once in the resulting linearised inheritance graph. For this reason, CLOS can be categorised in the linear multiple inheritance models. CLOS has been criticised for the unexpected or non-intuitive behaviour resulting from the linearisation algorithm (Baker, 1991).
- In mixin-based inheritance on the contrary, the programmer has explicit control over the linearisation: the inheritance chain is explicitated. This avoids unforeseen insertions of mixin-classes between a class and its parent: the resulting behaviour can hardly be considered unexpected or non-intuitive. Explicit control also allows the same mixin to occur more than once in an inheritance chain. The construction of the class **DoubleBoundedPoint** for instance will require the Bound mixin to appear twice and the Point mixin only once. It is exactly this peculiarity of mixins that opens the way to gain total expressiveness on duplication of attributes of common ancestors.

A direct consequence of this difference is that mixin-based inheritance cannot be categorised in any of the existing multiple inheritance models. Not in linear multiple inheritance, since mixins can occur more than once in the inheritance chain; nor in graph or tree based multiple inheritance, since each mixin can only have one direct parent. Strictly speaking mixin-based inheritance is a form of single inheritance. However, since one and the same mixin can be employed to construct different classes, the code reuse advantages of multiple inheritance are obtained.

4.2. Extending mixin-based inheritance to cover multiple inheritance

The mixin-based inheritance model proposed by Bracha & Cook (1990) treats all name collisions as specialisations and needs to be extended in order to obtain other conflict resolution strategies.

One could consider upgrading linear mixin-based inheritance to multiple mixin-based inheritance by providing a mixin with multiple parametrical super variables (as mixin-variant for class qualification). However such an approach would undo the linear property of mixins and would consequently reintroduce common ancestors and the associated duplication dilemma.

Bracha & Lindstrom (1992) start from a slightly different mixin model without a default conflict resolution strategy: all name collisions must be solved explicitly at mixin combination time. To this extent, a suite of mixin operators is provided including some specifically destined to deal with multiple inheritance name collisions. However, no complete analysis of their exact relation with multiple inheritance problems is given.

- A *rename* operator is proposed to obtain duplication. But whereas renaming is well-suited to deal with homonymous attributes, it is not the most appropriate way to deal with duplication in general. With renaming we *must* choose new names. The inadequacy of this obligation will become apparent later on in this text (section 5.3).
- To obtain sharing, a *restrict* operator is offered. This operator removes the definition of an attribute from a mixin. The attribute becomes virtual in the sense that internal references to it will be bound later on when the mixin is composed. When composing several mixins containing an homonymous attribute, the programmer can *arbitrarily* choose which definition she wants to retain by removing (i.e. restricting) all the other definitions. But, as we will illustrate in section 7.1, by grouping attributes in mixins a software designer constrains the possible classes that can be constructed with mixins. The restrict operator bypasses these design constraints. Therefore we consider it too powerful to serve as a basic solution for name conflicts.

The essential idea of our paper is opposite to the above suggestions. We will use an encapsulation mechanism to deal with multiple inheritance. Bracha & Lindstrom (1992) also introduced an encapsulation mechanism on

[1] This algorithm can be altered by meta-programming (using the CLOS meta-object protocol).

mixins, namely the *hide* operator. They did however *not* show how to use this operator with respect to multiple inheritance. On the contrary, they suggested to use the above mentioned *rename* and *restrict* operators for this purpose. We prefer encapsulation because it deals more appropriately with duplication and preserves the design behind mixins. Moreover, by using only one operator, all kinds of name collisions are treated in a uniform way. The uniformity of this solution reflects the similarity between the multiple inheritance problems of section 2.

Yet another reason why we find encapsulation so important is inherent to mixin-based inheritance. A mixin subclass is precisely intended to be applicable to different superclasses. Hence the need for encapsulation towards inheritors (as recognised in section 2.3) becomes the more important since it gives rise to a higher degree of substitutability of the superclass.

We will formally present a mixin-based model containing two operators. An incremental modification mechanism under the form of a mixin composition operator; and an encapsulation mechanism that is similar to the one of Bracha & Lindstrom, except for the treatment of super calls. Multiple inheritance name collisions either correspond to overriding, which will be handled by the composition operator, or to a collision where one attribute is not a specialisation of the other. In that case the encapsulation operator will solve the conflict. Because both operators will be defined in an orthogonal way, attribute duplication can be obtained by composing a mixin with itself in combination with encapsulation to solve the inevitably associated name collisions. This way we obtain a minimal multiple inheritance mechanism that uniformly captures the different interpretations of name collisions, as will be illustrated in section 5.

4.3. Super sends at method level

In this subsection we discuss a feature that is included in our model because it is considered to be good object-oriented programming practice.

Cook & Palsberg (1989) formally described mixins as wrapper functions of the form λself.λsuper.body. *self* and *super* may occur free in *body*, and are used to represent self-reference and the super-structure to be specialised respectively. Intuitively this means that a method in a mixin can refer to any method that is being defined in the super mixin. This is similar to the Smalltalk approach, where the super pseudo variable can be used to invoke any of the parent methods.

As already suggested by our special notation for super calls, we believe that a super call in a method should only be able to refer to the method it is overriding. As pointed out in section 3, Carré & Geib (1990) showed that message lookup with class qualification violates late binding and consequently restrains reusability. A super call to another attribute similarly redirects the method lookup mechanism, and consequently suffers from the same problem. For example, in Figure 6 the super call on the right wrongly prohibits overriding of *y*. A caller of an attribute should always allow an overridden version to be activated with his call. For this reason, super calls to other methods should be replaced by self sends. This way the new implementation of those methods is used whenever one of them becomes overridden, due to late binding of self.

Needing to invoke the super-part of another attribute instead of its new overridden implementation, indicates that this super-part should be represented by an extra attribute with a different name. This extra attribute can then be called with a self send.

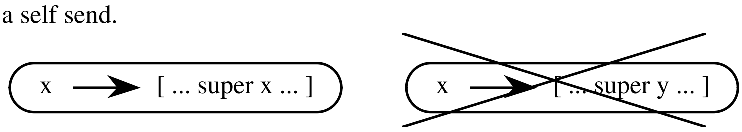

Figure 6: Allowed super calls

This informal discussion allows us to decide that, instead of parametrising subclasses with their parent class, methods should be parametrised with their super variant. In other words, the λ*super* should be moved from mixin level to method level. This makes a mixin a generator function of the form λself.record, where *record* is a record of method-bodies; and a method-body is a function of the form λsuper.body where *self* and *super* may occur free in *body*. The denotational semantics will be given more into detail in the following subsections.

This restriction on the use of super clearly avoids an overlap between the functionalities of self sends and super calls, and can consequently be considered as an illustration of the orthogonality principle for programming languages.

4.4. Formal definition of mixins

In this and the following subsection we formally present the syntax and denotational semantics of objects and mixins.

To make the examples more attractive we have included instance variables (in a slot-based way). In our formalism however, state is not explicitated: we will not give the denotational semantics of the accessor methods associated with instance variables. It would only make the model more difficult and lead us away from the essence of the problem. We want to define encapsulation and inheritance without stating anything about object identity.

As a consequence *imperative* statements, indicated in the examples between brackets and separated by semicolons, are not modelled. In our formalism a method body will only consist of a single *functional* expression.

4.4.1. Syntax

Below the grammar rules for the syntax of mixins are presented. The definition of objects will be given later.

Mixin	→	Method | Compose | Encapsulate
Method	→	Label "(" "#" Label ")" "=" Object
Compose	→	"(" Mixin "+" Mixin ")"
Encapsulate	→	"Encaps" "(" Mixin "," "{" Label "}" ")"
Label	→	Letter { Letter }
Letter	→	"a" | "b" | ... | "z"

Methods can make use of an argument, and return an object. Methods can be combined by using the following two operators on mixins: composition (+) and encapsulation (Encaps). Mixins are closed under these operators: the result of an operator on mixins is again a mixin that can further be used by both operators. We will see in the denotational semantics that the operators are orthogonal. Combinations of application of the operators cover a full-fledged class mechanism with multiple inheritance. Finally section 4.5.3 introduces an instantiation operator on mixins yielding a complete object-oriented system.

To make the examples more appealing, we will introduce the shorthand notation

Encaps(M, {a,b})   for   Encaps(Encaps(M, {a}), {b})

and similarly for more than two labels.

As an example of the syntax, the **SafeLamp** of section 2.3 could be written as follows:

Encaps(Lamp + SafeIntensity, {intensity, intensity()})

where Lamp and SafeIntensity are mixins, and intensity is the instance variable that needs to be encapsulated.[2]

4.4.2. Denotational Semantics of mixins

First we will give the semantic domains needed. The most important semantic domains represent mixins, objects and method-bodies:

- Mixins are functions expecting a self context and returning an object.
- Objects are modelled as records mapping labels onto method-bodies. Note that all these labels are visible to everyone. There is no need to consider non-public methods since we provide an explicit encapsulation mechanism.

[2] For reasons of simplicity, and since we have not formally defined instance variables, both acessor methods have the same label.

- Method-bodies are entities expecting a super variant, and performing a computation with an object as result. The method-body as well as its super variant can make use of an argument to implement their behaviour.

ArgRecord	**=**	**Label**	$\rightarrow$	**Object**
Mixin	**=**	**Object**	$\rightarrow$	**Object**
Object	**=**	**Label**	$\rightarrow$	**Methodbody**
Methodbody	**=**	**ArgObject**	$\rightarrow$	**ArgObject**
ArgObject	**=**	**Object**	$\rightarrow$	**Object**

Using these domains, the denotational semantics of mixins is given via the semantic function $\llbracket\ \rrbracket_m$.

$\llbracket$Mixin$\rrbracket_m$: **Mixin**		
□		
$\llbracket L_1(\#L_2)=O\rrbracket_m$	=	λself.{ $L_1 \rightarrow \lambda$super.λarg.$\llbracket O\rrbracket_o$ self {$L_2 \rightarrow$ arg} super }
$\llbracket M_1 + M_2\rrbracket_m$	=	$\llbracket M_1\rrbracket_m$ & $\llbracket M_2\rrbracket_m$
$\llbracket$Encaps(M,{L})$\rrbracket_m$	=	**encaps** ($\llbracket M\rrbracket_m$) L

where {key $\rightarrow$ val} is a record (as defined in Cook, 1989) with a single slot.

In the following two subsections, we will describe the semantics of the composition (&) and encapsulation (**encaps**) operator more into detail.

4.4.3. Composition operator

The semantic composition operator & corresponds to an incremental modification mechanism for mixins. It defines by means of record composition how a method is combined with its super variant.

& : **Mixin** $\rightarrow$ **Mixin** $\rightarrow$ **Mixin**

m_1 & m_2 = λself. [(m_1 self) $+_r$ λlabel.λsuper. m_2 self label (({label $\rightarrow$ λsuper.super} $+_r$ m_1 self) label super)]

where $+_r$ is the right preferential concatenation of records (as defined in Cook, 1989).

Super calls are regulated as follows: every occurrence of a super call in the method-body of a method in m_2 is replaced by the method-body of a homonymous method in m_1, while the super calls in m_1 remain to be filled in. This idea is similar to the definition of inheritance in Bracha & Cook (1990) except that super handling has been moved from mixin level to method level.

4.4.4. Encapsulation operator

The encapsulation operator defined below is a variant of the *hide* operator of Bracha & Lindstrom (1992). It makes us possible to encapsulate attributes in such a way that they become invisible to further clients.

Intuitively, encapsulating an attribute consists of removing its label from the domain of the mixin, after having replaced recursively all self sends to this attribute by the actual method-body (using a fixed point operator **Y**). Consequently these sends will never be redirected anymore.

encaps : **Mixin** $\rightarrow$ **Label** $\rightarrow$ **Mixin**

encaps m L **=** λself. [(**Y**(λfix.λself2. m (self2 $+_r$ { L $\rightarrow$ λsuper.λargs. fix self2 L $\perp$ args})) self) \ L]

where \ is a record restriction operator: in r \ L the attribute corresponding to label L is removed from the record r if it is present in r (as defined in Cook, 1989).

To avoid interference with the composition operator, only attributes with method-bodies performing no super calls are allowed to be encapsulated. This way encapsulated attributes cannot interfere with parent attributes. Similarly, since all self sends to an encapsulated attribute are replaced recursively, encapsulated attributes cannot interfere with child attributes. Encapsulation and composition can therefore be considered orthogonal.

This orthogonality improves the comprehensibility of our model and prohibits unusual constructions not belonging to common object oriented ideas. Consider for example the parent and child mixin in Figure 7. Attribute *a* in ChildMixin performs a super call. Encapsulating it and composing the resulting mixin with ParentMixin results in a strange behaviour of the super call: the super call in *a* has become a kind of aggregation because the self send *a* (remember that receiverless messages denote self sends) in the parent is not redirected to the *a* version of the child since this version is hidden to any client. Such an overlap between aggregation (commonly obtained by message passing) and inheritance is avoided. As in most object-oriented systems inheritance and aggregation are strictly separated software design concepts.

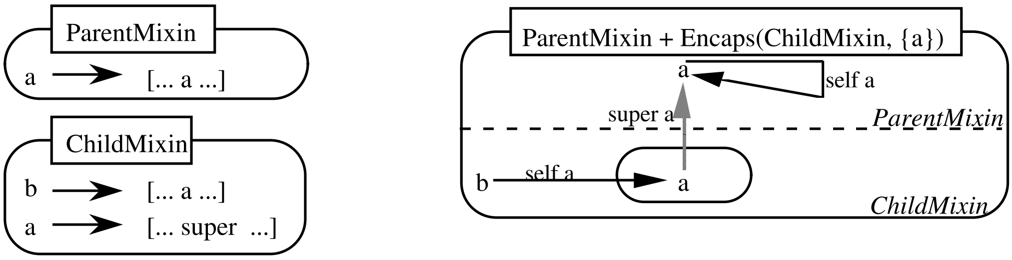

Figure 7: Encapsulating an attribute that performs a super call.

As mentioned by Cook (1989), other record composition operators than $+_r$ (corresponding to a different conflict resolution strategy) can be used in the definition of mixin composition. Thanks to the orthogonality, the definition of our composition operator can be altered independently of the other mixin operators.

4.5. Formal definition of objects

Now we present the syntax and denotational semantics of objects. Because our model is stateless, objects have no identity. This way they rather correspond to object values than to full fledged objects.

4.5.1. Syntax

Object	→	Instantiate | ObjectSend | SelfSend | SuperCall | ArgRef | Primitive | ε
Instantiate	→	"(" Mixin ")" "new"
ObjectSend	→	Object Label Argument
SelfSend	→	Label Argument
SuperCall	→	"super" Argument
Argument	→	"(" Object ")"
ArgRef	→	"#" Label

ε denotes the empty object. super and new are reserved keywords. new is the syntactical representation of the instantiation operator that will be discussed in section 4.5.3. It is only included for reasons of completeness.

Self sends, super calls and ordinary message sends can make use of an argument. The formal argument names (ArgRef) are preceded by a # symbol to avoid confusion with receiverless self sends. In this syntax, only one argument at a time can be passed with message sends and super calls. Multiple arguments can be added in a straightforward way, but this would unnecessarily clutter the formal model. When no arguments are provided, we adopt the convention to omit the parentheses.

An object can be a primitive object like a number or a string. Since this is not essential to our model, their syntax and denotational semantics will not be explicitated.

4.5.2. Denotational Semantics of objects

$\triangleright$Object $\rrbracket_o$ :: **Object**	→ **ArgRecord** → **ArgObject** → **Object**	
□		
$\triangleright$(M)new $\rrbracket_o$	=	λself.λargs.λsuper. **inst** ($\triangleright$M $\rrbracket_m$)
$\triangleright$(O_1)L(O_2) $\rrbracket_o$	=	λself.λargs.λsuper. **send** ($\triangleright O_1 \rrbracket_o$ self args super) L ($\triangleright O_2 \rrbracket_o$ self args super)
$\triangleright$#L $\rrbracket_o$	=	λself.λargs.λsuper. args L
$\triangleright$L(O) $\rrbracket_o$	=	λself.λargs.λsuper. **send** self L ($\triangleright$O $\rrbracket_o$ self args super)
$\triangleright$super(O) $\rrbracket_o$	=	λself.λargs.λsuper. super ($\triangleright$O $\rrbracket_o$ self args super)
$\triangleright$ε $\rrbracket_o$	=	λself.λargs.λsuper. {}

As can be seen in the semantic domains given earlier, methods can make use of the super variant of the method body they override. {} denotes the empty record.

To send a message to an object, the auxiliary function **send** is needed. Performing a self send occurs by sending a message to the self object.

send :	**Object** → **Label** → **Object** → **Object**
send receiver L arg =	receiver L ⊥ arg

4.5.3. Instantiation Operator

In classical class-based languages, classes with methods that call as yet undefined attributes cannot be instantiated. They are called "abstract superclasses": their only reason of existence is to be inherited from. Similarly, in mixin-based languages not every mixin can be instantiated. However, things are a bit different here. A mixin that contains self sends to attributes that it does not define itself, can not only be used as ancestor to inherit from, but also as descendant relying on definitions of parent mixins. Therefore we cannot speak of abstract supermixins, nor of abstract submixins. We simply call a non-instantiable mixin an *abstract mixin*[3]. Another reason for a mixin to be abstract is the presence of a method containing a super call. We will not formalise the definition of abstract mixin since this is not so important for this paper.

inst :	**Mixin** → **Object**
inst m =	*if* m *is abstract then* ⊥ *else* **Y**(m)

In this definition we can see that instantiation of a mixin (that is not abstract) creates an object (or instance) in which all self references are permanently bound to the object itself. Since this usage of self gives rise to inherent recursive behaviour, we need a fixed point operator (as in the definition of the encapsulation operator) for instantiation. This approach is similar to the one followed by Cook & Palsberg (1989).

5. Solution of the multiple inheritance problems

Now that we have been more specific about our formal mixin model, we illustrate how the problems outlined in section 2 can be solved in a uniform way, and why an explicit encapsulation mechanism goes hand in hand with mixin-based inheritance.

5.1. Encapsulation of attributes

Reconsider the coloured lamp example of section 2.3. As mentioned earlier, the code components in Figure 5 represent mixins. Initially, mixins consist of public attributes only. The line separating public and non-public attributes in the mixins of Figure 5 should no longer be taken into consideration since encapsulation has become explicit. If we want a lamp class with instances that cannot directly access *intensity*, we should encapsulate *intensity*:

Encaps(Lamp, {intensity(), intensity})

[3] Using the terminology of Bracha & Cook (1990), an "abstract" (resp. "instantiable") mixin is called "partial" (resp. "complete").

A SafeLamp class[4] without directly accessible intensity can be constructed as follows:

SafeLamp = Encaps(Lamp + SafeIntensity, {intensity(), intensity})

In other words, we first need to compose Lamp with the SafeIntensity mixin that overrides the *intensity* attribute, and then we have to make this attribute invisible by encapsulating it, as illustrated in Figure 8.

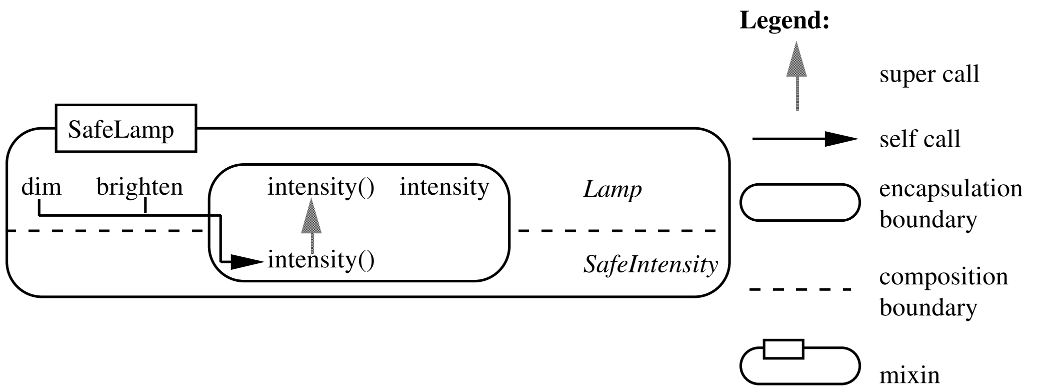

Figure 8: Safe lamp mixin with encapsulated intensity

In agreement with the definition of our encapsulation operator, we can deduce the following scope rules in Figure 8:

-a self send pointer can only enter or exit an encapsulation boundary in its own horizontal level (between composition boundaries)

-a super call can never cross an encapsulation boundary

The encapsulation of *intensity* in SafeLamp can also serve inheriting clients. For example, the previously defined SafeLamp mixin can be extended with the Colour mixin to obtain a ColouredSafeLamp, schematically represented in Figure 9.

ColouredSafeLamp = SafeLamp + Encaps(Colour, {intensity(), intensity})

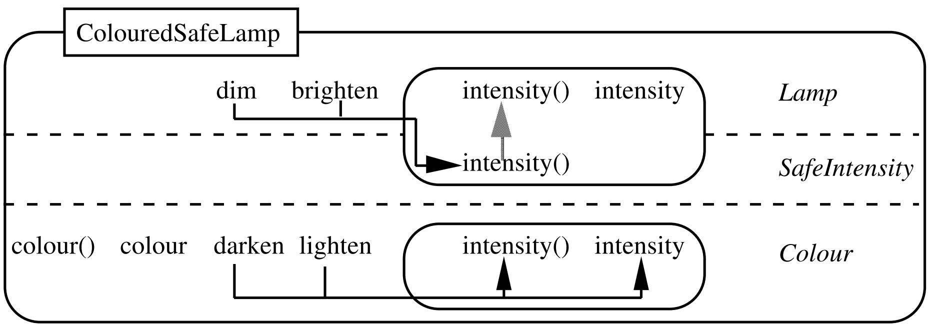

Figure 9: Avoiding unintended name collisions through encapsulation

These examples illustrate that attributes should not be defined directly as e.g. "public", "protected" or "private", but that encapsulation should be arranged at mixin using level. Moreover, the same encapsulation mechanism should be used for all clients. This allows us to provide a whole range of different interfaces, all constructed with one and the same encapsulation mechanism. For instance the implementor of SafeLamp receives the Lamp mixin, whereas the implementor of ColouredLamp and message passing clients get a lamp mixin with encapsulated *intensity*.

[4] We will no longer put class names in bold because classes are now mixins just as the code components earlier.

Notice that encapsulation does not distribute over composition. For example, the class ColouredLamp of section 2.3 can be constructed as follows:

ColouredLamp = Encaps(Lamp, {intensity(), intensity}) + Encaps(Colour, {intensity(), intensity})

But in the following construction the brightness and colour intensities would be mixed up, since the methods *dim* and *brighten* will act on the colour intensity that overrides the brightness intensity!

WrongLamp = Encaps(Lamp + Colour, {intensity(), intensity})

5.2. Sharing and duplication of attributes in common ancestors

Let 's reconsider the point example of section 2.1. The constant upper bound functionality where *bound* is hidden towards all clients, is represented by the following abstract mixin:

ConstantBound = Encaps(Bound, {bound})

ConstantBound is abstract since its *move* method performs an unbound super call. It can be used as stand-alone extension for different classes, illustrating the multiple inheritance power of mixins. For example a bounded point class with visible *x* and *y* instance variables is constructed by:

XYBoundedPoint = Point + ConstantBound

To prevent clients from directly accessing *x* and *y* , we need to encapsulate these attributes:

BoundedPoint = Encaps(XYBoundedPoint , {x,x(),y,y()})

Suppose that we add a printed history to our bounded points, i.e. we want to build the class HistoryBoundedPoint. This cannot be done by composing BoundedPoint with History, because History needs access to the *x* and *y* attributes while BoundedPoint encapsulated them. For this reason we have to use XYBoundedPoint:

HistoryBoundedPoint = Encaps(XYBoundedPoint + History, {x(),x,y(),y})
= Encaps(Point + ConstantBound + History, {x(),x,y(),y})

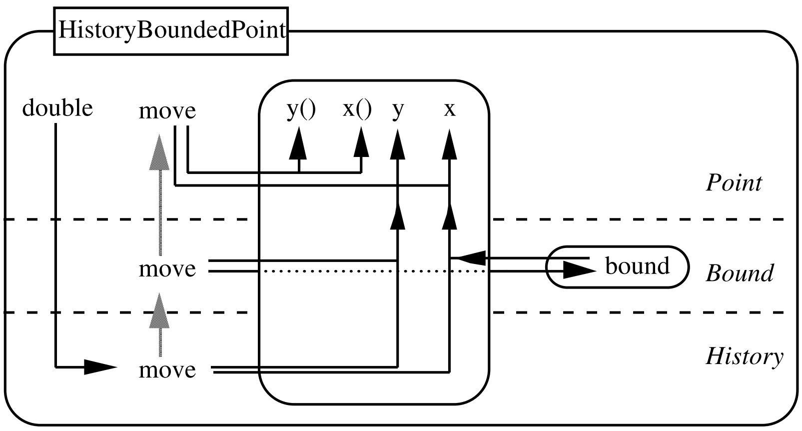

Figure 10: Bound and History share the Point attributes

HistoryBoundedPoint is represented in Figure 10. The Bound mixin and the History mixin both share the attributes of the Point parent. We can also see that late binding of self sends is achieved, because the *double* method always refers to the most recent *move* method.

The relative order in which the bound and the history functionality are added determines the behaviour represented by the resulting class. Consider:

BoundedHistoryPoint = Encaps(Point + History + ConstantBound, {x(),x,y(),y})

BoundedHistoryPoint is very similar to HistoryBoundedPoint, except that the last two arguments for the + operator have been reversed. As a result both mixins have a different behaviour. An instance of HistoryBoundedPoint will record the current position even when the point is tried to be moved outside the boundary, as opposed to a BoundedHistoryPoint that only prints a history of successful invocations of the *move* method. Consequently the composition operator is not commutative.

The sinus bound facility, as required by the class SinusBoundedPoint with encapsulated *x* and *y* is obtained as follows:

SinusBound = Encaps(Bound + Sinus, {bound})

SinusBoundedPoint = Encaps(Point + SinusBound, {x(),x,y(),y})

Now we can show that duplication of attributes in common ancestors is obtained by composing with the same mixin twice. Recall that the DoubleBoundedPoint class combined both the constant and the sinus bound and therefor needed duplication of the Bound attributes.

DoubleBoundedPoint
= Encaps(XYBoundedPoint + SinusBound,{x(),x,y(),y})
= Encaps(Point + ConstantBound + SinusBound,{x(),x,y(),y})
= Encaps(Point + Encaps(Bound,{bound}) + Encaps(Bound+Sinus,{bound}), {x(),x,y(),y})

The last lines are added to illustrate that Bound is applied twice, once specialised with Sinus.

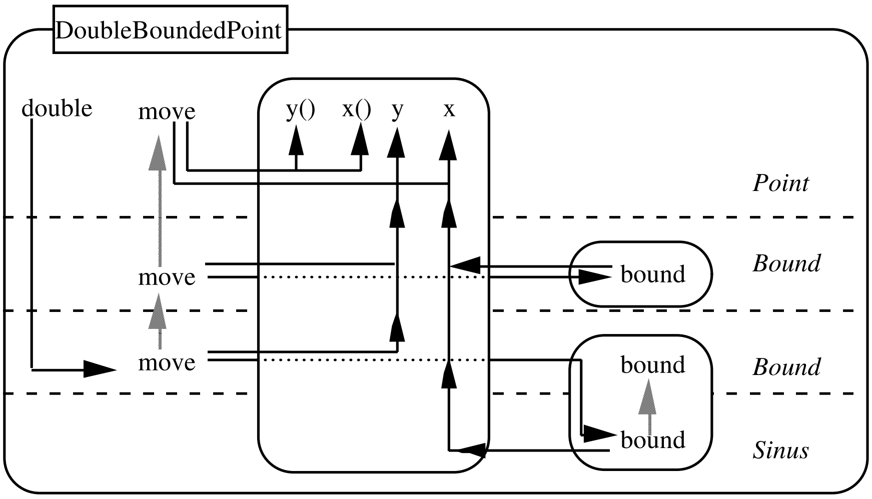

Figure 11: Schematic representation of the DoubleBoundedPoint mixin

We can conclude that multiple application of a mixin, in this case Bound, together with separate encapsulation of some of its attributes, in this case *bound*, results in duplication of these attributes. More specifically multiple versions of these attributes are retained. This can be seen in Figure 11. The *move* method has not been encapsulated. Only one version is held that is invoked twice (via successive super calls).

The examples in this subsection illustrated that the combination of encapsulation and composition provides the required flexibility concerning sharing and duplication. Note that the mixin components of section 2.1 do *not* intend to cover all possible kinds of bounded points. For example, an OrBoundedPoint in which the *y*-value of the point is required to stay below the constant bound *or* the sinus bound cannot be created, since Bound is designed to impose one additional bound. Repeatedly applying it logically results in an *and*-combination of the bounds. Our approach intends to use mixins only in constructing the behaviour they are designed for, or to put it another way, to respect the design behind mixins. We will say more about mixin design issues in section 7.

5.3. Homonymous attributes

The last remaining problem concerns homonymous attributes in different superclasses, as explained in section 2.2. In fact the *bound* attributes of the subclasses SinusBoundedPoint and BoundedPoint can be seen as homonymous attributes too. It is rather coincidental that both bounds stem from a common ancestor. The difference between the duplication problem and the homonymous attributes problem concerns visibility towards subclasses. In the point example, both versions of the *bound* attribute were invisible to the subclass DoubleBoundedPoint. In the person example however, we want the subclass SportsmanWithReduction to be able to refer to both *cardNr* attributes. Since these attributes should be distinguishable we need to rename them. Renaming doesn't require an extra operator but can be obtained by using the mixins in Figure 12.

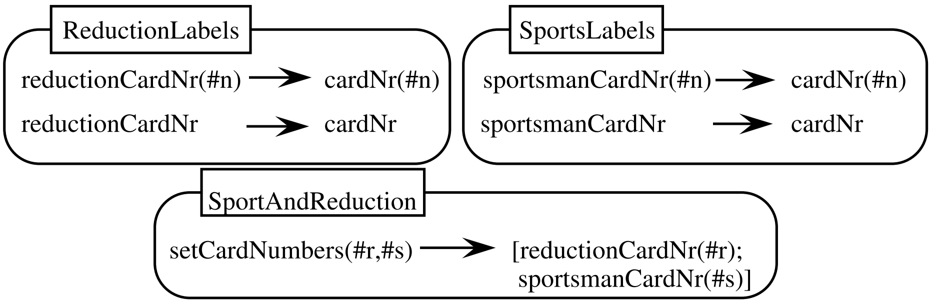

Figure 12: Renaming and extension mixins

Making SportsmanWithReduction is then simply done by composing ReductionCard with ReductionLabels and SportsmanCard with SportsLabels, followed by encapsulating the card numbers:

```
SportsmanWithReduction =
    Person +
    Encaps(    Encaps(ReductionCard + ReductionLabels, {cardNr,cardNr()}) +
               Encaps(SportsmanCard + SportsLabels, {cardNr,cardNr()}) +
               SportAndReduction,
             {reductionCardNr(),sportsmanCardNr()} )
```

This solution does not suffer from the reusability restrictions caused by class qualified message passing. Indeed, the renaming mixins can also be composed with specialisations of card mixins, for instance that for TopSportsmanCard (see also Figure 13):

```
TopSportsmanWithReduction =
    Person +
    Encaps(    Encaps(ReductionCard + ReductionLabels, {cardNr,cardNr()}) +
               Encaps(SportsmanCard + TopSportsmanCard + SportsLabels,{ cardNr,cardNr()}) +
               SportAndReduction,
             {reductionCardNr(),sportsmanCardNr()} )
```

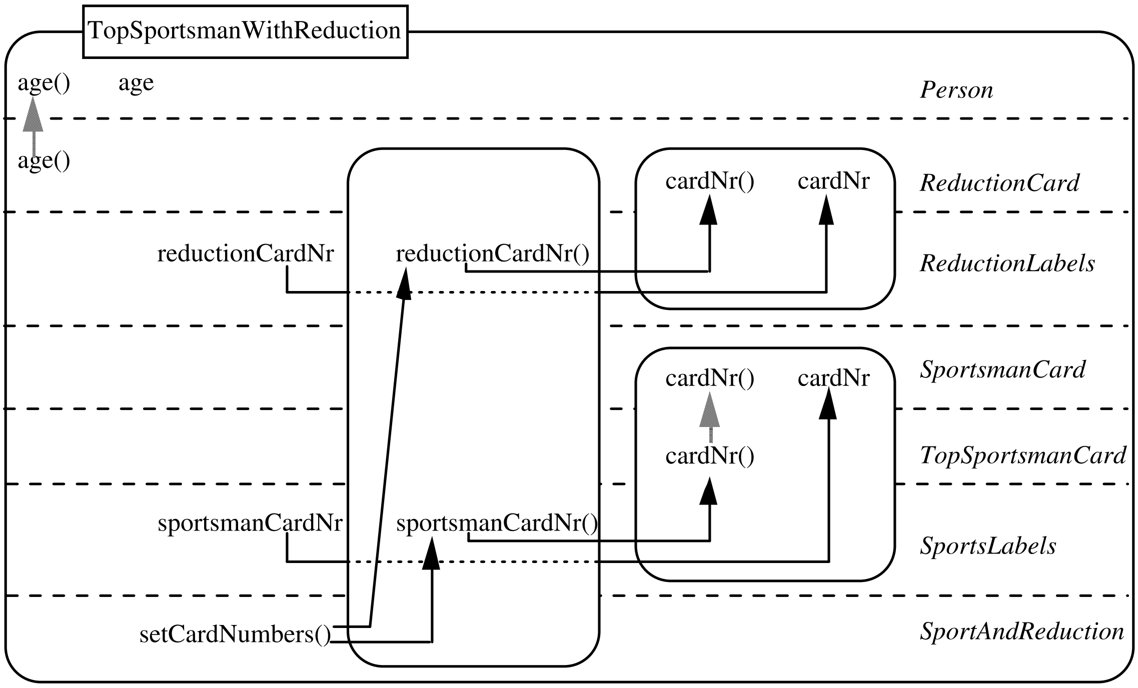

Figure 13: Duplication and renaming of the card numbers

The nuisance of writing and combining the auxiliary mixins ReductionLabels and SportsLabels can be eliminated by introducing a rename operator as syntactic sugar. This approach is different from other approaches like Eiffel that introduce renaming as an extra semantic construct. But as mentioned in section 3, Eiffel's renaming is too tightly coupled to the inheritance mechanism. Bracha & Lindstrom (1992) also offer a renaming operation at semantic level. As opposed to Eiffel, their rename operator on mixins is orthogonal with inheritance. Nevertheless it is not fully suited as a means to deal with multiple inheritance for the following reasons:

- Renaming doesn't provide encapsulation. Consequently, encapsulation is needed anyway. Moreover we have seen in the example how to model a kind of renaming via encapsulation.

- Although renaming can solve the homonymous attributes problem, sometimes there is no need to explicitly rename the homonymous attributes. This is the case when the subclass specialisation code does not need to refer to the homonymous attributes. DoubleBoundedPoint for example does not need to access its different bounds simultaneously. For this reason, encapsulation of the bound attributes again seems a better alternative, since it relieves us from the obligation to choose different names. In an environment of dynamic inheritance, this annoying obligation can become a defect since a mixin could be applied a statically unknown number of times. In such a case it is impossible to choose different names.

- Bracha's rename operator is more powerful than our renaming strategy but it seems that this additional power not really strengthens multiple inheritance. A rename operator is rather a tool to integrate independently-developed components as in Hölzle (1993). This exceeds "pure" multiple inheritance where the interface is part of the agreement between the developers of different components. Using Bracha's rename operator, SportsmanWithReduction would be constructed as follows:

```
SportsmanWithReduction = Person +
Rename(ReductionCard, [cardNr→reductionCardNr, cardNr()→reductionCardNr()]) +
Rename(SportsmanCard, [cardNr→sportsmanCardNr, cardNr()→sportsmanCardNr()])
```

  The additional power of renaming lies in the possibility to convert SportsmanWithReduction to TopSportsmanWithReduction *afterwards*:

TopSportsmanWithReduction = SportsmanWithReduction +
Rename(TopSportsmanCard, [cardNr→sportsmanCardNr, cardNr()→sportsmanCardNr()])

If there had been a self send to *cardNr* from within the mixin SportsmanCard, this self send would be redirected to the top sportsman version. If on the contrary the attribute is renamed via encapsulation, as we propose, that self send will not be redirected. But such a redefinition, with the intention to redirect the self call, necessarily relies on the internal structure of SportsmanWithReduction. Consequently such a redefinition can as well be mixed in directly after SportsmanCard, just as we mixed in TopSportsmanCard.

5.4. Classical class-based versus mixin-based approach

This subsection shows that uncoupling inheritance and encapsulation would be less powerful in a traditional class-based approach than in a mixin-based formalism. Because class extensions in conventional class-based inheritance are not stand-alone, encapsulation cannot be limited to the extension only. This restricts the combination of composition and encapsulation to the following constructions:

Encaps(Encaps(Encaps(Encaps(Parent + Extension1) + Extension2) + Extension3) + Extension4)

The following adaptation to our point example illustrates the necessity for encapsulation of the extension separately. Imagine that we asked somebody to implement a MaxStep mixin that limits the distance a point can be moved in one step. The instance variables that hold the maximal step size coincidentally have been named *x* and *y,* as in Figure 14.

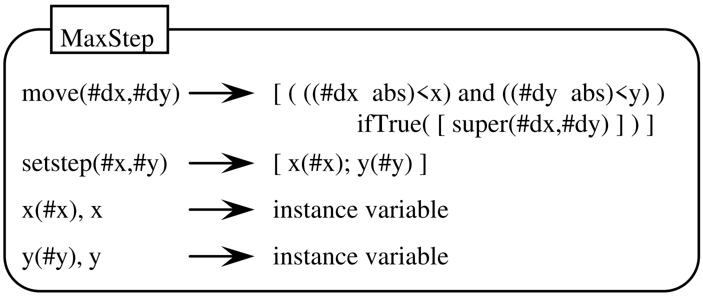

Figure 14: A mixin limiting the step size

We now want to construct a class that represents points with limited step size and with recorded history. More specifically we want these points to record their current position for every attempt to move, even when the point is not moved due to a too big step size. Since Point also has instance variables named *x* and *y*, the name conflict has to be resolved. The described behaviour is obtained as follows:

XYHistoryStepPoint = Point + Encaps(MaxStep, {x,x(),y,y()}) + History

The encapsulation has to be limited to the MaxStep extension alone. We can not encapsulate *x* and *y* in Point since History needs to access them. In other words we must be able to make attributes local to sub-mixins (i.e. extensions). Note that in the above example reversing the order of the composition operands doesn't work either since it results in another behaviour.

We can conclude that the stand-alone characteristic of mixins raises explicit encapsulation to its full import. Mixin-based inheritance allows to separately encapsulate attributes in the extension and in the parent.

6. Linearisation criticised

Besides encapsulation, linearisation is a key feature in our solution of the duplication dilemma. Systems that linearise the inheritance hierarchy - including mixin-based inheritance - inherently only treat one direct parent, i.e. only one super pseudo variable.[5] The power of mixin-based inheritance stems from the fact that this single super variable can be filled in with different parents.

6.1. Rejection of criticisms on linearisation

Snyder (1987) stresses that the use of inheritance in a software component should not be exposed to clients, in order to be able to reimplement the parent with another inheritance hierarchy. On the other hand our discussion about duplication and sharing of attributes stresses the need for a more global view on the inheritance graph to obtain the necessary inheritance expressiveness. Stand-alone mixins appear to be a good compromise between both contradictory requirements. Composition of mixins results in a new mixin, establishing a hierarchical exposure of inheritance structure. While building a mixin, one is exposed to and can consequently appropriately alter the component chain it is built of; once the mixin is created the component chain becomes invisible.

Another criticism is that linearisation forces an ordering between immediate superclasses. Section 5.2 however showed that BoundedHistoryPoint and HistoryBoundedPoint represent another behaviour. Consequently two subclasses that multiply inherit from BoundedPoint and HistoryPoint are needed, even at analysis level. Instead of considering the semantically significant order of both direct parents as a flaw, we appreciate it as a mechanism to specify the difference between both subclasses.

A related criticism is that a forced ordering of superclasses prevents a compiler to choose an optimised order (Baker, 1991). If the order in which some mixins are composed with another mixin is irrelevant (always resulting in the same behaviour), one could consider to introduce an additional syntactical construct, permitting the programmer to add at once a (unordered) set of mixins. Such a construct then suggests the compiler to search for an optimised order.

6.2 Limitation of our model because of linearisation

As mentioned above linearisation implies the restriction to one super variable. Consequently our multiple inheritance mechanism is not capable to implement the notion of views. Consider the example of a Person with a *name*, a Professor that inherits from Person by overriding the existing *name* with a new one that adds a prefix "Prof.", and a Doctor that overrides *name* by adding a prefix "Dr." (see Figure 15).

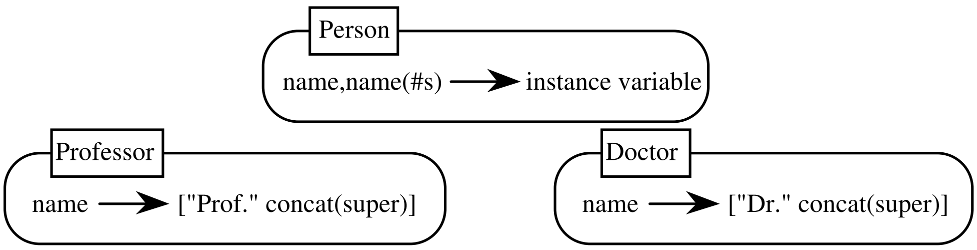

Figure 15: Person-, Professor- and Doctor-mixins

A person that is both professor and doctor, is represented by:

```
ProfessorDoctor = Person + Doctor + Professor
```

Because both Professor and Doctor share the Person name in a linearised inheritance chain, this has the effect of adding as prefix "Prof. Dr.".

There is however another kind of behaviour that, starting from the mixins of Figure 15, cannot be expressed with our model. Suppose we want to create a professor-doctor with two views: one only as professor, and one only as doctor. Invoking *name* on the professor-view should only concatenate "Prof.", while *name* in the doctor-view should only concatenate "Dr.". As opposed to the person example in section 5.3 where a

[5] The super variable can refer to the whole parent, or to only one method of the parent (as in our case). This aspect is however irrelevant to the current discussion.

sportsman with reduction has two card numbers, a professor-doctor should only have one name, shared in the Person parent. It appears to be impossible to use the Professor and Doctor mixins of Figure 15 in combination with a renaming mechanism similar to section 5.3 for constructing the two views. We cannot encapsulate two overriding variants of the same *shared* attribute separately.

With two super variables, one for professor and one for doctor, on can imagine a solution. But this way, as mentioned before, the duplication dilemma for *name* in Person is re-introduced. Moreover it is debatable if an inheritance mechanism must provide the means to implement views or that the notion of views should be added orthogonally to an inheritance mechanism.

7. Extensions towards reliability

The principle of orthogonality is severely violated in many class-based languages. It is widely understood that the class concept is heavily overworked. Among others, classes are used for incremental modification, classification, determining attribute visibility and typing. Bracha & Lindstrom (1992) enumerate no less than 11 different roles classes play. We are currently working on orthogonally enhancing our basic model with independent features in order to obtain a level of static reasoning comparable to that of classical class-based systems, yet with a larger flexibility. We will briefly discuss three of them.

7.1. Mixin normalisation

Until now we have grouped attributes together in mixins on a rather intuitive basis. Grouping attributes together in mixins should however not be an arbitrary choice. Together with Sakkinen (1989), we believe that the distribution of attributes over mixins is a class design issue that should be guided by introducing a kind of normalisation mechanism for mixins (Van Limberghen, 1995), similar to normalisation in relational databases. Grouping attributes can be exploited as a controlled restriction on the possible classes that can be constructed. We now give three examples of such restrictions.

- When composing two mixins, all the attributes of the second one override or are added to the first. By putting different attributes together in the same mixin, the designer decides that it is for example impossible to compose two mixins in such a way that some of the attributes of the first mixin override those of the second, while other attributes in the second mixin override those of the first. An example of this kind of behaviour is shown in Figure 16.

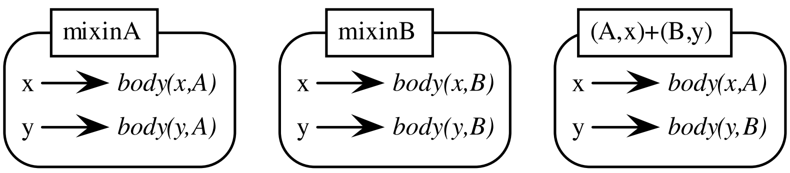

Figure 16: Attribute dependent composition

  By putting the attributes *x* and *y* together this way in mixinA and mixinB, the designer explicitly excluded (A,x)+(B,y) as representative for an element of his universe of discourse.

- A second example involves multiple invocation of method bodies that perform a super call. By putting *x* and *y* together in mixin A (Figure 17), the mixin designer took the decision to exclude the construction of classes where, upon sending x and y respectively, *body(x,A)* and *body(y,A)* are invoked a different number of times. For example, in the class A+B+C both *body(x,A)* and *body(y,A)* are invoked only once. In A+B+A+C *body(x,A)* and *body(y,A)* are invoked twice. *x* or *y* can be specialised in B or C, as illustrated in Figure 17.

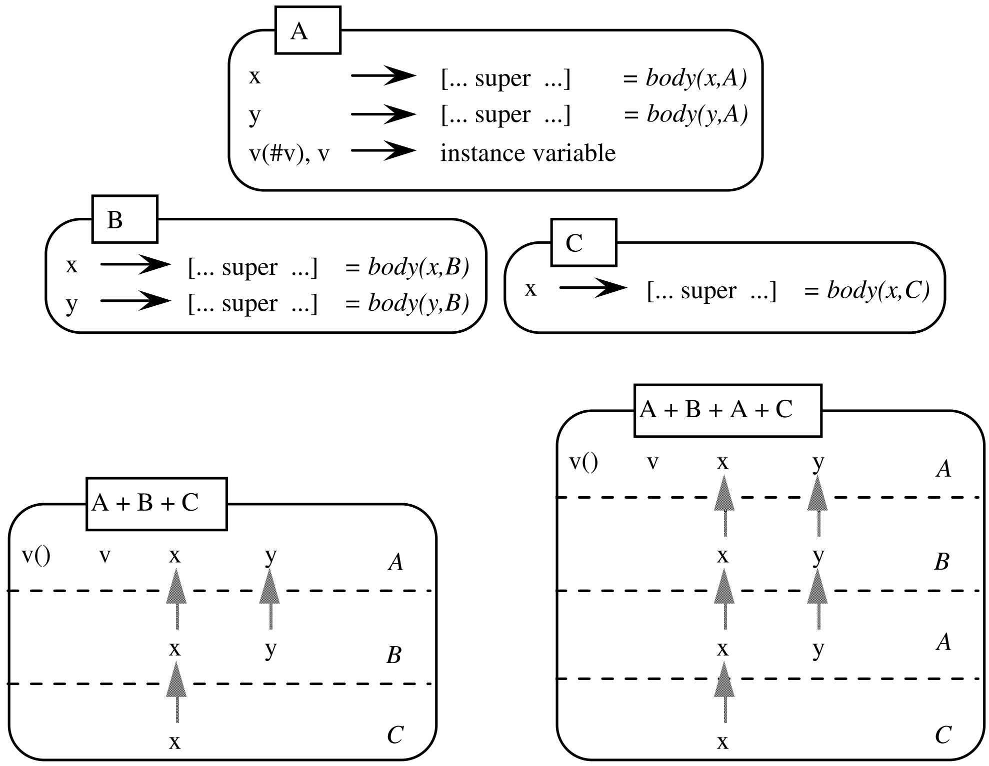

Figure 17: *body(x,A)* and *body(y,A)* are executed the same number of times

The class A+B+A+C contains only one version of the instance variable *v*. *v* is shared while *body(x,A)* and *body(y,A)* are duplicated. In Sakkinen (1989) this situation is called “fork-join inheritance”. Sakkinen claims that “subobject integrity” can be violated in the sense that updating shared attributes and accessing duplicated attributes of A may cause unexpected side effects between the B and the C part of the object. These side effects are indeed unexpected if both B and C are inherited from, ignorant of the existence of the common ancestor A. In our model on the contrary, we consider the possible side effects to be intended, because of the explicitness of the inheritance chain and because *v* is visible. If *v* is encapsulated, as for instance in EncBPart + EncCPart where

EncBPart = Encaps(A+B, {v(),v}) and EncCPart = Encaps(A+C, {v(),v})

then *v* will be duplicated. This way unexpected side effects are avoided. Therefore our approach supports subobject integrity more strongly than other approaches to fork-join inheritance.

- A last example of the relation between the universe of discourse and attribute grouping is the ReductionCard mixin of section 2.2, in which seemingly different attributes as *age* and *cardNr* are grouped together to force owners of a reduction card to be younger than 25 years.

Bracha & Lindstrom (1992) started from the opposite standpoint. They offer mixin operators to construct *any* kind of behaviour starting from a given set of mixins. Their *restrict* operator removes (the definition of) an attribute from a mixin. This way attribute grouping is disrupted. It even becomes possible to convert a grouping of attributes in mixins into any other grouping. As a result, grouping attributes in mixins no longer has any significance. An encapsulation operator on the contrary does not disrupt attribute grouping: it does not remove an attribute from a mixin, but simply hides it.

7.2. Mixin classification

Mixins are chunks of code that can be freely combined using the offered operators, allowing unanticipated combinations of behaviour to be made. If uncontrolled, one faces an explosion of possible combinations of mixins. A mechanism to control this combinatorial explosion is needed.

In traditional class-based systems the class hierarchy partially fulfils this combination restricting role, but still has to be enriched with extra restricting capabilities. Multiple inheritance is less expressive than it appears, essentially in its lack to put constraints on multiple inheritance from different classes. For example we should be able to prevent a class to inherit from the classes Male and Female simultaneously. To this extent, Hamer et al. (1992) include *classifiers* in their class hierarchy. We are thinking about a similar classification mechanism especially destined for mixins, preserving its characteristic of parametrical super binding.

7.3 Typing

As opposed to many current languages, subclassing and subtyping should be separated (Canning et al., 1989). Therefore inheritance should only be seen as a subclassing, and not as a subtyping mechanism. In this paper instances of a subclass are not necessarily substitutable for instances of the superclass. A type system for mixin-based inheritance similar to the one described by Lucas et al. (1995) can be added orthogonally to our model.

8. Conclusion

Multiply inheriting from different classes raises the question of how to treat shared ancestors. It should be possible to share some ancestor attributes while duplicating the others. Many current multiple inheritance systems however lack to do so. Secondly, homonymous attributes inherited from different parents are often dealt with by label renaming or qualifying labels with class names. Both solutions are criticised because they restrain reusability.

Mixin-based inheritance constitutes the basis for the required expressiveness on the aspect of sharing and duplication, but ignores different interpretations for multiple inheritance name collisions. We shed a new light on multiple inheritance by taking encapsulation into account. We pointed out how to obtain all sorts of name conflict strategies using mixin-based inheritance enhanced with an explicit encapsulation operator. Reusability was hereby preserved since label renaming or class qualifying was not needed. This way we obtained a minimal model that uniformly captures the different interpretations of name collisions. Consequently this model seems to constitute a good basis for OO languages that include multiple inheritance in their scope.

As already shown in (Bracha & Lindstrom, 1992) viewing inheritance as a composition of software components opens the way for new OO software methodologies. Mixin-based inheritance makes the parent-child relationship much more symmetrical than conventional inheritance. While classical class hierarchies only consist of classes that can be subclassed, it now becomes possible to offer mixins as stand-alone extensions where the parent remains to be filled in.

Mere mixin-based inheritance exhibits combinatorial flexibility of software components but lacks conceptual meaning. In order to obtain a level of static reasoning comparable to that of classical class-based systems, we are currently working on mixin design, amongst others mixin classification and normalisation. We criticised the *restrict* operator of Bracha & Lindstrom (1992) because it counteracts these design issues. An encapsulation mechanism on the contrary respects mixin design. Therefore we consider it as a valuable software engineering extension to pure mixin-based inheritance.

9. Acknowledgements

We express our gratitude to our promotor Theo D'Hondt, and to Thomas Kühne, Carine Lucas and Patrick Steyaert for their useful comments on earlier versions of this paper. We also thank Niels Boyen, Wolfgang De Meuter and Kim Mens for taking a closer look at the formal aspects of our approach. Last but not least, we are indebted to Russel Winder for guiding us to acceptance of the paper and to the anonymous referees, for their interesting suggestions and remarks.

References

Baker,H. G (1991), CLOStrophobia: Its Etiology and treatment, OOPS Messenger 2(4), ACM Press, pp. 4-15.

Bracha, G & Lindstrom, G (1992), Modularity meets Inheritance, in Proc. of International Conference on Computer Languages, 1992, IEEE Computer Society, pp. 282-290. Also available as Technical report UUCS-91-017.

Bracha, G & Cook, W (1990), Mixin-based Inheritance, in Meyrowitz, N, joint OOPSLA/ECOOP ’90 Conference Proceedings, ACM Press, pp. 303-311.

Borning, A H & Ingalls, D H (1982), Multiple Inheritance in Smalltalk 80, Proceedings at the National Conference on AI '82, pp. 234-237.

Canning, W R, Cook, W L, Hill, W L, Olthoff, W G (1989), Interfaces for Strongly-Typed Object-Oriented Programming, in Meyrowitz, N, OOPSLA '89 Conference Proceedings, ACM Press, pp. 457-467.

Cardelli, L (1988), A Semantics of Multiple Inheritance, Information and Computation 76, Academic Press, pp. 138-164.

Carré, B & Geib, J (1990), The Point of View notion for Multiple Inheritance, in Meyrowitz, N, joint OOPSLA/ECOOP ’90 Conference Proceedings, ACM Press, pp. 312-321.

Cook, W (1989), A Denotational Semantics of Inheritance, PhD thesis, Department of Computer Science, Brown University.

Cook, W & Palsberg, J (1989), A Denotational Semantics of Inheritance and its Correctness, in Meyrowitz, N, OOPSLA '89 Conference Proceedings, ACM Press, pp. 433-444.

Ellis, M & Stroustrup, B (1991), The Annotated C++ Reference Manual, Addison-Wesley.

Hamer, J, Hosking, J G & Mugridge W B (1992), Static Subclass Constraints and Dynamic Class Membership Using Classifiers, Technical Report, University of Auckland, Computer Science Department.

Hense, A V (1992), Denotational Semantics of an Object-oriented Programming Language with Explicit Wrappers, Formal Aspects of Computing 3, pp. 1-27.

Hölzle, U (1993), Integrating Independently-Developed Components in Object-Oriented Languages, in Nierstrasz, O, ECOOP '93 Conference Proceedings, Springer-Verlag, pp. 36-56.

Keene, S E (1989), Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS, Addison-Wesley.Knudsen, J (1988), Name Collision in Multiple Classification Hierarchies, in Gjessing, S & Nygaard, K, ECOOP ‘88 Conference Proceedings, Springer-Verlag, pp. 93-109.

Lucas, C, Mens, K, Steyaert, P, "Typing Dynamic Inheritance. A Trade-Off between Substitutability and Extensibility", Technical Report vub-prog-tr-95-03, Vrije Universiteit Brussel, Department of Computer Science.

Meyer, B (1988), Object Oriented Software Construction, Prentice Hall.

Moon, D A (1986), Object-oriented programming with Flavors, in Meyrowitz, N, OOPSLA '86 Conference Proceedings, ACM Press, pp. 1-8.

Van Limberghen, M (1995), Normalising class components, Technical Report vub-prog-tr-95-05, Vrije Universiteit Brussel, Department of Computer Science.

Sakkinen, M (1989), Disciplined Inheritance, ECOOP '89 Conference Proceedings, Springer-Verlag, pp. 39-56.

Snyder, A (1987), Inheritance and the Development of Encapsulated Software Components, Shriver, B & Wegner, P, Research Directions in Object-Oriented Programming, MIT Press, pp. 165-188.

Ungar, D & Smith, R (1987), SELF: The power of simplicity, in Meyrowitz, N, OOPSLA '87 Conference Proceedings, ACM Press, pp. 227-242.