Vrije Universiteit Brussel
Faculteit Wetenschappen

A Marriage of Class- and Object-Based Inheritance Without Unwanted Children

Patrick Steyaert   Wolfgang De Meuter

Techreport vub-prog-tr-95-02

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

A Marriage of Class- and Object-Based Inheritance Without Unwanted Children

Patrick Steyaert   Wolfgang De Meuter

Free University of Brussels
Programming Technology Lab (PROG)
Pleinlaan 2 - 1050 Brussels - Belgium
email: {prsteyae, wdmeuter}@vnet3.vub.ac.be

Abstract:

Class-based languages have often been criticised for their rigidity. Prototype-based languages on the other hand are usually considered too flexible. We claim that this flexibility is inherently correlated to encapsulation problems. We therefore propose a new scale of "encapsulated" inheritance mechanisms on objects that combine the object model of class-based languages with the inheritance mechanism of prototype-based languages. It is shown that they combine the advantages but shun the disadvantages of both paradigms. A denotational semantics of an exemplar encapsulated inheritance mechanism on objects is given and compared with the standard denotational semantics of class- and object-based inheritance. This comparison confirms our claims.

1. Introduction

In the evolution of object-oriented programming two, largely disjoint, groups of languages have emerged: *class-based languages* and *prototype-based languages*. Both include mature and well established languages: the first group includes languages such as Smalltalk and C++, the second one includes languages like Self.

Languages fitting neatly in this taxonomy are mostly characterised by their use or non-use of classes as a way to express incremental modification. Class-based languages define inheritance on classes while prototype-based languages do this on objects. As argued by Lieberman, this is a reflection of the more philosophical controversy between representing general concepts as abstract sets (or classes) and representing concepts as prototypes (see [Lieberman86]).

We agree with [Stein&al.89] that this controversy cannot be decided in one or the other way and that much can be gained from combining the strengths of both viewpoints.

The reaction of *hybrid languages* (such as Hybrid) is to allow both classes and objects to be modified and to offer operators to turn classes into objects (instantiation) and vice versa (promotion) (see [Stein87]). Hybrid languages limit themselves to a straightforward *union* of the distinctive language features of both paradigms thereby combining their advantages but, more importantly, also their disadvantages (figure 1).

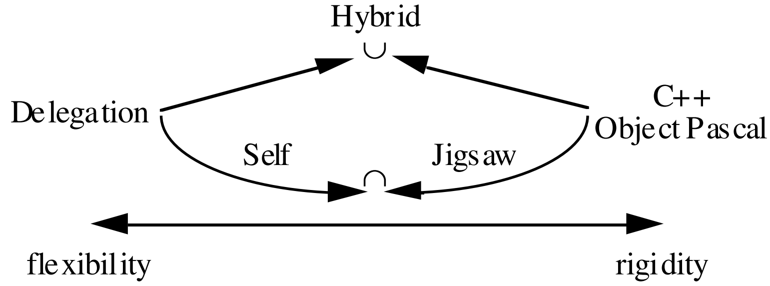


Figure 1

More recently, both the prototype- and class-based language research communities have created languages that strive toward an *intersection* of the paradigms (figure 1). Examples of these are attempts to introduce more rigidity into prototype-based languages giving rise to object-based inheritance[1] (see [Dony&al.92] and [Stein&al.89]). In the other school, language mechanisms such as mixin-based inheritance and modular inheritance (see [Bracha&Cook90] and [Bracha92]) are an improvement for class-based languages as they allow a more flexible creation of class hierarchies. Neither improvement is entirely successful. The latter languages stay in the realm of class-based languages as they still promote a distinction between modifiable templates and non-modifiable objects. On the other hand, there is a general feeling that object-based inheritance is still too flexible. In this paper we will identify the exact reason upon which this feeling is based. We will show that current object-based inheritance languages suffer from an inherent encapsulation problem and that this problem must be solved for them to become viable alternatives to class-based languages.

In order to give sufficient rigour to our arguments we will analyse these problems and propose solutions in a denotational framework. By comparing denotational semantics we will demonstrate that the encapsulation problems are inherently present in object-based inheritance and inherently absent in pure class-based inheritance. The lessons learned in the class-based world will then be used to propose a new family of encapsulated inheritance mechanisms on objects. Finally, we will discuss how these new mechanisms can provide the basis for languages in a successful intersection of classes and prototypes.

2. Terminology and Concepts

In this section we will briefly introduce the basic concepts and semantic domains needed in the remainder of the paper. The object model that is used in the paper is introduced in section 2.1. This is the fairly common treatment of objects modelled as records. Readers unfamiliar with specialisation interfaces should read section 2.2.

[1] The behaviour sharing mechanism of prototype-based languages varies from delegation to object-based inheritance. With object-based inheritance parents are determined at object-creation time while delegation usually allows an object to change parents dynamically.

Section 2.3 introduces the semantic domains for classes and inheritance on classes in the same way as [Cook&Palsberg89]. Section 2.4 introduces our denotational treatment of object-based inheritance. It also briefly compares the semantics of class- and object based inheritance.

2.1. Objects and Client Interfaces

In the current state of the art, no general definition of objects is widely accepted. Still a consensus exists that *encapsulation* is one of the distinguishing features of objects (see [Mens&al.94] [Nierstrasz92] [Snyder86] [Steyaert94] [Wegner87]). An object has an interface which contains the messages it understands and which hides its implementation details. This interface is usually referred to as the object's *client interface*. A record listing all the operations belonging to the client interface of an object thus serves as a natural model for it. Denotationally, objects are therefore modelled as

$$\text{Object} = \text{Ident} \rightarrow \text{Attribute}$$

In this model, message passing is a combination of attribute selection and invocation. Commonly, attributes are methods. Attribute invocation is then a simple procedure invocation thereby passing the actual arguments to the method. In order to respect encapsulation, message passing must be an atomic operation, i.e. selection and invocation must be part of an indivisible whole.

As we will show, existing object-oriented programming languages (OOPLs) not always make it possible to model their 'objects' by the above equation. This is particularly true for prototype-based languages. We will therefore use the term *object* as a general noun to indicate all possible language constructs that can act as receivers of messages. Whenever we want to make a clear distinction we will use the term *encapsulated objects* to indicate ‘objects’ that can be modelled by the equation and that allow atomic message passing.

2.2 Incremental Modification and Specialisation Interfaces

The power of object-oriented programming lies in its provision for so-called incremental programming techniques. One creates a new program construct R (a 'result') by specifying its differential M (a 'modification') with respect to some existing program construct P (a 'parent') (see [Wegner&Zdonik88]).

Inheritance is a specific kind of incremental modification. First of all it is characterised by the fact that modifiers can invoke operations in their parent. Thus the result must be modelled as $R = P + M(P)$. A second characteristic is that modifiers can override operations of the parent with 'late binding of self' which indicates that any self reference made by the parent P will, in the result R, refer to the entire R instead of just P. The semantics of 'late binding of self' will be discussed in section 2.3.

Inheritance requires modifiers to know which parent operations they can invoke and what the effect will be of overriding attributes. This is illustrated in the following example (figure 2). A 'Person' is defined whose client interface contains two operations ('print' and 'getName'). The 'print' operation in 'Person' invokes the 'getName' operation in its implementation. This knowledge is used by the modifying 'Female' to assure that overriding the 'getName' method also has an effect on the result of its (inherited) 'print' method. In general, a parent exposes this kind of knowledge through a second interface, called the *specialisation interface*.

```
Person (name)
        method getName is return name
        method print is self.getName.print

Female Modifies Person
        method getName is return "Ms." concat(super.getName)

AlternativePerson (name)
        method getName is return name
        method print is name.print
```

Figure 2

In this example the specialisation interfaces are implicit: they can only be derived by inspecting the implementations. The issue of specifying specialisation interfaces is outside the scope of this paper (see [Lamping93], [Kiczales&Lamping92]). This however, does not prevent us to reason about them. Consider figure 2 again. Although specialisation interfaces are left implicit, one can still observe that the 'Female' modification clearly does not fit 'AlternativePerson' with the same result. This is because 'Person' and 'AlternativePerson' have different specialisation interfaces.

Note that the specialisation interface *exposes implementation details* to modifiers. In the above example only a minimal set of implementation details are exported (self and super interactions). Practical programming languages often expose much more to modifiers. For example, most languages also disclose the private variables of a parent through the specialisation interface leading to a form of decapsulated inheritance. Different forms of decapsulation (exposure) of implementation details for modifiers have been discussed in the literature (see [Snyder86]).

2.3. Classes

The most popular incremental modification mechanism in OOPLs is inheritance on classes. A class plays at least two different roles: one as a template for the instances that will be created from it, and one as a parent for further refinement by inheritors.

This section explains the denotational counterpart of these roles, which is mainly due to [Cook&Palsberg89].

2.3.1 Classes as Templates

In its most basic form classes are used as templates to create objects with a common behaviour. Denotationally, classes are therefore modelled as *generators of objects* and *instantiation* is an operation that transforms a generator into an object:

$$\text{Instantiate} : \text{Generator} \rightarrow \text{Object}$$

$$\text{Generator} = \ldots \rightarrow \text{Object}$$

In this equation the domain "…" from which objects are generated is left open. In OOPLs where objects can refer to themselves, objects are typically modelled as self referential records. This can be abstractly formulated as: self = record (self). Using fixed point analysis, self = record (self) is then rewritten as self =(λs. record(s))self and thus self = Fix(λs. record(s)). Hence, λs. record(s) is a generator function for objects and the exact type of generator functions is given by:

$$\text{Generator}_{\text{Fix}} = \text{Object} \rightarrow \text{Object}$$

Generator functions are used to model classes and instantiation is achieved by taking the fixed point of a generator. The generator domain allows fixing: it is a function of which the argument domain is the same as the result domain.

$$\text{Instantiate} : \text{Generator}_{\text{Fix}} \rightarrow \text{Object} \ \text{ where Instantiate}(g) = \text{Fix}(g)$$

2.3.2 Class-based Inheritance

As shown by Cook and Palsberg, the model of classes based on generator functions is well suited for extension with inheritance. Generators can be combined with late binding of self to form subclasses. Since subclasses should be potential new parents, inheritance should be an operation converting two generators into a new generator whose self is distributed over all its subparts. This is captured in the 'Inherit'-operator

$$\text{Inherit}_{\text{Simple}} : \text{Generator}_{\text{Fix}} \rightarrow \text{Generator}_{\text{Fix}} \rightarrow \text{Generator}_{\text{Fix}}$$

$$\text{Inherit}_{\text{Simple}} = \lambda P.\ \lambda M.\ \left(\lambda \text{self}_{\text{object}}.\ (P\ \text{self}_{\text{object}}) +_r (M\ \text{self}_{\text{object}})\right)$$

where $+_r$ is the right-preferential record combinator.

This model gets slightly more complicated when modifiers are allowed to invoke parent operations (Smalltalk's super-calls). Therefore, modifiers need to be parameterised by a super object. Generators that are parameterised by super objects are called "*wrappers*" (see [Cook&Palsberg89]).

$$\text{Wrapper}_{\text{Fix}} = \text{Object} \rightarrow \text{Generator}_{\text{Fix}} = \text{Object} \rightarrow \text{Object} \rightarrow \text{Object}$$

The inheritance operator essentially stays the same. Besides distributing a new 'self' (for late binding) to all subparts, it also provides the modifier M with a reference to the parent.

$$\text{Inherit}_{\text{Cook\&Palsberg}} : \text{Generator}_{\text{Fix}} \rightarrow \text{Wrapper}_{\text{Fix}} \rightarrow \text{Generator}_{\text{Fix}}$$

$$\text{Inherit}_{\text{Cook\&Palsberg}} = \lambda P.\ \lambda M.\ \left(\lambda \text{self}_{\text{object}}\ .\ (P\ \text{self}_{\text{object}}) +_{\text{r}} (M\ (P\ \text{self}_{\text{object}})\ \text{self}_{\text{object}})\right)$$

2.4. Object-Based inheritance

Whereas class-based inheritance involves incremental modification of classes, object-based inheritance does the same on objects. This is illustrated in the following example where objects are modified such that their name is printed with a "Ms." prefix.

```
Rita = Object (name="Rita")
            method getName is return name
            method print is self.getName.print
Linda = Object (name="Linda")
            method getName is return name
            method print is self.getName.print
method MakeLetterHead ( aPerson )
            return aPerson extended with
                method getName is return "Ms. ".concat super.getName
LetterHeadRita = MakeLetterHead (Rita)
LetterHeadLinda = MakeLetterHead (Linda)
```

Figure 3

Object-based inheritance also involves late binding of self but on the level of objects instead of classes. The mechanism of class-based inheritance is therefore entirely shifted to the object level. This requires objects to be modelled by some kind of generators. Because the receiver ('self') can also be subject to extension and objects are represented as generators, the interpretation of 'self' should also be a generator. Hence, conventional generators and fixed points will not do the job properly as objects need an unfixed rather than a fixed version of themselves for self-reference. This requires ordinary self-application instead of fixed points. We will call this operation 'Wrap'.

$$\text{Wrap} : \text{Generator}_{\text{Wrap}} \rightarrow \text{Object}$$

$$\text{Wrap} = \lambda g\ .\ (g\ g)$$

The modified generator and wrapper domains are given below. In contrast with the previous generator domain, the new domain allows self-application: its argument domain is of the same type as the generator domain itself.

$$\text{Generator}_{\text{Wrap}} = \text{Generator}_{\text{Wrap}} \rightarrow \text{Object}$$

$$\text{Wrapper}_{\text{Wrap}} = \text{Object} \rightarrow \text{Generator}_{\text{Wrap}}$$

The quite subtle difference between wrapping and taking a fixed point is illustrated in figure 4.

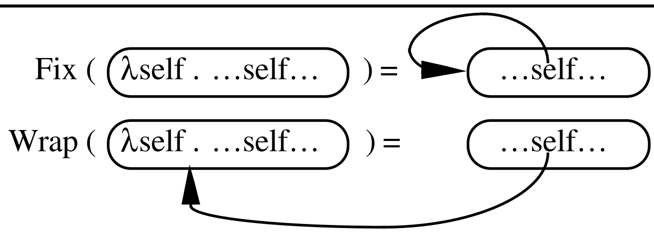


Figure 4

Taking the fixed point of a function offers the function an internal version of itself. Wrapping a function on the other hand, gives the function an external version of itself; i.e. it becomes possible for the function to refer to itself as if it were neither wrapped nor fixed.

Two important remarks have to be made here.

First, the previous inheritance *technique* remains applicable but, of course, on the new generator domain.

$$\text{Inherit}_{\text{Objects}} : \text{Generator}_{\text{Wrap}} \rightarrow \text{Wrapper}_{\text{Wrap}} \rightarrow \text{Generator}_{\text{Wrap}}$$

$$\text{Inherit}_{\text{Objects}} = \lambda P.\ \lambda M\ .\ (\lambda \text{self}_{\text{generator}}\ .\ (P\ \text{self}_{\text{generator}}) +_{r} (M\ (P\ \text{self}_{\text{generator}})\ \text{self}_{\text{generator}})\ )$$

From this it can be concluded that *class-based and object-based inheritance share the same inheritance mechanism*. This can be regarded as the semantic counterpart of an insight discovered by Stein who has shown that it is perfectly possible to program 'prototype-style' in a class-based language when classes are treated as objects (see [Stein87]).

Second, the consequence of representing objects as generators is that message passing must always be preceded by wrapping the receiver as the receiver should be provided with a correct self-reference each time a message is sent. This is also the major difference with class-based inheritance. *Class-based and object-based inheritance have a different message passing mechanism and therefor a different object model.*

3. Object-Based Inheritance Breaches Encapsulation

The main objection against conventional object-based inheritance is that it suffers from an inherent encapsulation problem with respect to specialisation interfaces. This needs some explanation.

In the literature much has been said already about breaching of encapsulation by inheritors. Snyder, for example, discusses how different inheritance strategies allow inheritors to breach the encapsulation of the parent [Snyder86]. The most simple form of decapsulation is where an inheritor can directly access the instance variables of its ancestors.

The direct consequence of breaching encapsulation is that a parent of which the encapsulation is violated, *cannot* freely decide to change its implementation: all encapsulation breaching inheritors need to change their implementation with the parent. The solution of Snyder is to introduce encapsulated inheritance: inheriting clients have no direct access to the private attributes of their parent.

This does not prevent all violation of encapsulation however. Late binding of self, which is inherent to inheritance, also involves a form of decapsulation. Take for example a class-based encoding of our person example (figure 2). In this example the female subclass breaks the encapsulation of its person parent class by overriding the getName method, i.e. the person class can not change its implementation without having an effect on the implementation of the female subclass. If, for example, the person class decides to implement its print method in the style of the alternative person class (figure 2) then the female class will need to adapt its implementation also. So, although encapsulation towards inheritors is an important issue, it is clear that inheritors need to breach the encapsulation of their parent class to a certain degree. This is what makes inheritors special. *Not allowing* inheritors to use the specialisation interface of their parent would be a total neglect of the power of inheritance. *Allowing* to use the specialisation interface is a violation of encapsulation.

In class-based languages this is not a serious problem since the inheritance relation is rather static. The set of classes that breach the encapsulation of a parent is limited to a statically known set of inheritors and a clear distinction is made between inheriting and other clients. For object-based inheritance this is not the case.

```
CircleWithExpensiveGoldenWindow is Object
        private variable ExpensiveGoldenWindow
        method DrawInWindow (aWindow)
            ...
        method Draw
            self.DrawInWindow(ExpensiveGoldenWindow)

WindowThief is Object
        method StealGoldenWindow (aCircle)
            is return (aCircle extended with
                                 override method DrawInWindow(aWindow)
                                             is return aWindow
                      ).Draw ; -- this actually does NO drawing

WindowThief.StealGoldenWindow(CircleWithExpensiveGoldenWindow)
```

Figure 5

Conventional object-based inheritance allows any object to be extended by any other. Therefore, anyone who can access the client interface of an object can also see its specialisation interface. In other words, with respect to the object involved, no distinction can be made between clients and inheritors. All clients can thus rely on the implementation details exported by the specialisation interface. This can even be stated more strongly. Object-based inheritance makes it impossible for a programmer to specify an *interface* (in the literal sense of the word). A client, dissatisfied with the restrictions an client interface imposes, can always patch up some additional methods in order to enhance its access priorities. This is certainly the case in object-based languages where inheritors have direct access to parent variables. But even in absence of this feature (so called “encapsulated inheritance” [Snyder86]) client interface encapsulation can be breached. This is illustrated in the example in figure 5. By a clever overriding of circle methods, a window thief is constructed that exports the private acquaintance of circle objects to all possible clients.

This breaching of encapsulation towards clients is unacceptable. This is restated in the following design principle:

The Immaculate Client Interface Principle for Objects

An object should not expose any other interface but the client interface to its message passing clients. It must be able to hide its specialisation interface from message passing clients.

Thus, concerning the client/specialisation interfaces a clear distinction must be made between message passing clients and inheriting clients. That object-based inheritance violates this principle is also reflected in its formal semantics. Indeed, in the previous section we showed that object-based inheritance is only possible if objects are modelled as generators instead of encapsulated objects (in order to ensure late binding of self). In such a model, however, any client is allowed to fill in the self parameter of an object at its own will. In other words, the very model of object-based inheritance requires that any client can always pass a new extension as the self parameter of an object. But then a permanent danger exists that a 'dummy'-extension is made with the intention of decapsulating the implementation details of an acquaintance.

Hence, at the semantic level, the encapsulation problems boil down to the fact that languages providing object-based inheritance confuse their objects with explicit language-representations of generators and generator-manipulating operators. Notice that Smalltalk somehow suffers from the same problems. Since everything in Smalltalk is an object (even classes and meta classes), at some point, the implementation of the `subclass:modification` method must take its own generator in order to mix it with the given modifier (read: wrapper). Hence, programming ‘prototype-style’ in Smalltalk by only making use of classes and omitting instantiation (as was illustrated in [Stein87]) suffers precisely from the same problem. Again this stems from the fact that at some point, deep down in Smalltalk, a generator and a generator manipulating function is used.

Respecting our principle can therefore be expressed in the semantics of OOPLs with the following criterion.

The Object Criterion

> An OOPL in which objects can be semantically modelled as encapsulated objects and that has no provisions for explicit generator manipulation is guaranteed to respect the immaculate client interface principle.

Current prototype-based languages do not pass the criterion and certainly do not respect the principle.

As can be deduced from the semantics outlined in section 2.3, the objects of pure class-based languages like Object-Pascal, C++ or Jigsaw (see [Bracha92]) can be modelled as encapsulated objects since they are fixed at instantiation time. Furthermore, pure class-based languages allow no explicit generator manipulation as their classes are compile-time structures. Hence, pure class-based languages pass the criterion and seem to be an adequate solution. However, pure class-based languages are a very drastic solution to the problems. They make a clear cut distinction between inheritable classes and non-inheritable objects. Specialisation interfaces are associated only with generators and not with objects. *Objects are not subject to extension* and therefore have no associated specialisation interface.

4. Prototypes With Encapsulated Specialisation Interfaces

Rather than concluding that pure class-based languages are the only solution, we will reconsider the design characteristics of object-based inheritance keeping in mind the need for a clear distinction between inheriting and other clients. A straightforward restriction of the visibility of specialisation interfaces to a limited set of 'inheritor' objects must be approached with caution however. As long as objects can *arbitrarily* subscribe to a specialisation interface, the same set of already discussed problems pops up. Solutions where the object under extension is *not* an active participant in the subscription process are *no* substantial improvements. In the next two sections an inheritance mechanism on objects is proposed that avoids these pitfalls. The result will be an evolutionary step towards an intersection of both paradigms (as indicated in figure 1).

4.1. The General Idea

Respecting the principle does not necessarily preclude inheritance from objects. On the contrary, even within the constraints, a whole scale of inheritance mechanisms remains possible. These mechanisms are characterised by the fact that they encapsulate the specialisation interface. Accordingly, at the semantic level they are characterised by the fact that generators are encapsulated, which is achieved by a hygienic use of wrapped generators to denote objects. Wrapped generators are encapsulated objects and therefore pass the first part of the criterion.

But the semantics of inheritance requires generators. Respecting the second part of the criterion thus implies that inheritance should be an operation of which the mechanics is hidden within the object. Extending an object is then accomplished through its client interface. We call this *encapsulated inheritance on objects*. So, the essence of our solution is that an object itself only provides a client interface but encapsulates a non encapsulated version of itself.

As this might seem a bit fuzzy, let us first give an example.

4.2. An Example Mechanism

The most stable version of encapsulated inheritance on objects that we have at the time of writing is an inheritance mechanism based on *mixin-methods.* Mixin methods were introduced in [Steyaert&al.93] and [Lucas&Steyaert94].

As indicated above, inheritance is an incremental modification mechanism where a modifier M modifies a parent P in order to produce a result R: R = P + M(P). The insight of [Hense92] and [Bracha&Cook90] is to upgrade these modifiers to explicit language constructs. In language design they are called ‘mixins’ as they allow a modification to be mixed into a parent. Mixins are an attempt to improve the reusability of code.

The principle of mixin-methods is quite simple. Next to the usual methods, an object can list a number of mixin-methods in its interface. The definition of a mixin-method consists of variable declarations, method declarations and other mixin-methods. Mixin methods can be invoked —like ordinary methods— by sending a message. Upon invocation an extension of the receiving object is returned. This extension contains the declarations that can be found in the definition of the invoked mixin-method. Mixin methods were integrated in our Agora language (see [Codenie&al.94]).

The power of this particular form of encapsulated inheritance will now be shown by a kernel version of Agora, called MiniMix. The very simple syntax[1] of MiniMix is given by the following grammar rules[2].

P → OE

A program is an object expression

OE → ME **|** I **|** B **|** “**self**” **|** ”**super**” **|** ”**object**” AB

An object expression is (the result of) a message expression, an identifier, a basic object, a pseudo variable or an object created ex-nihilo with an abstraction.

ME → ”**send**” OE_1 I ”(” OE_2 ”)”

A message I is sent to an object OE_1 with OE_2 as an actual parameter

AB → ”**[**” AD ”**]**”

[1] See [Schmidt88] for an excellent introduction to formal syntax and semantics of programming languages.

[2] In the paper only the important aspects of MiniMix will be explained. The interested reader can find a complete formal semantics of MiniMix in the appendix.

$$\text{AD} \rightarrow \text{"}\mathbf{method}\text{"}\ I_1 \text{"("}\ I_2 \text{")"}\ \text{OE} \mid \text{"}\mathbf{mixin}\text{"}\ I_1 \text{"("}\ I_2 \text{")"}\ \text{AB} \mid \text{AD}_1 \text{"}\mathbf{;}\text{"}\ \text{AD}_2$$

Abstractions are sequences of methods and mixin-methods. They are used to create objects with. Mixins and methods are attribute declarations (AD). They both require a parameter I_2. The result of a method is the evaluation of its body OE while the result of a mixin-method is the object extended with the listed (mixin) methods, i.e. with a given abstraction (AB).

The syntax of the language and its use should be clear by the following MiniMix version of figure 2.

```
object [ mixin Person (name)
          [ method getName (void) name;
            method print (void) send send self getName(void) print(void);
            mixin MakeFemale (void)
               [ method getname (void)
                         send "Ms." concat ( send super getName(void))
               ]
          ];
         mixin AlternativePerson (name)
          [ method getName (void) name;
            method print (void) send name print(void);
            mixin MakeFemale (void)
               [ method getName (void)
                         send "Ms." concat( send super getName(void));
                 method print (void) send send self getName(void) print(void)
               ]
         ]
       ]
```

Figure 6

Let us now take a look at the semantics in order to explain why mixin-methods do pass the object criterion and therefore follow the immaculate client interface principle. For the sake of clarity, the semantic domains of MiniMix are summarised below[1].

$$\text{Object} = \text{Ident} \rightarrow \text{Attribute}$$

$$\text{Attribute} = \text{Object} \rightarrow \text{Object}$$

$$\text{Generator} = \text{Generator} \rightarrow \text{Object}$$

$$\text{Env} = \text{Ident} \rightarrow \text{Object}$$

$$\text{Wrapper} = \text{Object} \rightarrow \text{Generator}$$

An object expression is always evaluated using the current unwrapped self ('g' in the equations below), the current environment ('e' in the equations) and the current super object ('p' in the equations). The interpretations of identifiers and pseudo variables then are

[1] We will only consider attributes that are one-argument operations (i.e. Attribute = Object → Object).

$\mathbb{OE}$: **ObjExpr** $\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator $\rightarrow$ Object

$\mathbb{OE}$ [[**self**]] = λe . λp . λg . (Wrap g)

$\mathbb{OE}$ [[**super**]] = λe . λp . λg . p

$\mathbb{OE}$ [[I]] = λe . λp . λg . (e I)

The interpretation of self confirms that objects are modelled by encapsulated objects. A self reference is achieved by wrapping the 'current generator'.

The semantics of message passing consists of selecting the appropriate attribute in the interpretation of the receiver and applying this attribute to the semantic object denoting the argument:

$\mathbb{ME}$: **Msg** $\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator $\rightarrow$ Object

$\mathbb{ME}$ [[**send** OE_1 I (OE_2)]] = λe.λp.λg.(($\mathbb{OE}$[[OE_1]] e p g) I) ($\mathbb{OE}$[[OE_2]] e p g)

As can be seen in the grammar, an object can be created ex-nihilo by writing an abstraction **object** AB. Therefore, the semantics of ex-nihilo created objects requires understanding the semantics of abstractions. The body of a mixin-method is also an abstraction. Since the body of a mixin-method is a modifier, its denotation is a wrapper and hence abstractions are denoted by wrappers.

$\mathbb{AB}$: **Abstr** $\rightarrow$ Env $\rightarrow$ Wrapper

$\mathbb{AB}$ [[**[**AD**]**]] = λe.($\mathbb{AD}$ [[AD]] e)

An ex-nihilo created (encapsulated!) object is nothing more than a wrapped wrapper with an empty super ($\perp$[1]).

$\mathbb{OE}$ [[**object** AB]] = λe . λp . λg . (Wrap ($\mathbb{AB}$ [[AB]] e $\perp$))

An abstraction is a sequence of attribute declarations. As above, all attribute declarations are evaluated making use of the current environment, the current super and the current unwrapped self. This gives us the specification for the semantic function of attribute declarations.

$\mathbb{AD}$: **AttDec** $\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator $\rightarrow$ Object

A list of attributes is recursively defined by the concatenation of two such other lists. The same therefore goes for its semantics.

$\mathbb{AD}$ [[AD_1 **;** AD_2]] = λe.λp.λg.($\mathbb{AD}$ [[AD_1]] e p g) $+_r$ ($\mathbb{AD}$ [[AD_2]] e p g)

[1] All error handling and undefined values are handled by non-termination as these are not relevant to the paper.

At the lowest level in such a sequence, methods and mixin-methods are declared. This declaration binds a name to a one parameter function containing the effect of invoking the attribute (records are represented as [I→λo.…]).

$$\mathbb{AD}[\![\ \textbf{method}\ I_1\ (I_2)\ OE\ ]\!] = \lambda e.\lambda p.\lambda g.\Big[\ I_1 \rightarrow \lambda o.\mathbb{OE}[\![\ OE\ ]\!]\ e[I_2/o]\ p\ g\ \Big]$$

$$\mathbb{AD}[\![\ \textbf{mixin}\, I_1\ (I_2)\ AB\ ]\!] = \lambda e.\lambda p.\lambda g.\Big[I_1 \rightarrow \lambda o.\big(\text{Wrap}\ \big(\text{Inherit}_{\text{Objects}}\ g\ \ (\mathbb{AB}[\![\ AB\ ]\!]\ e[I_2/o])\big)\big)\Big]$$

In the case of methods the effect is the denotation of the method body expression in the current environment extended with the actual parameter binding. The effect of a mixin invocation is explained by extending the current unwrapped self with the wrapper denoting the mixin-method body $\big(\text{Wrap}\ \big(\text{Inherit}_{\text{Objects}}\ g\ \ (\mathbb{AB}[\![\ AB\ ]\!]\ e[I_2/o])\big)\big)$.

Mixin methods pass the object criterion. By a case analysis of the above semantic clauses, it can be verified that we make a hygienic use of wrapping each time an object is created: ex-nihilo, by invocation of a mixin-method and by returning a self object. Message passing thus involves neither wrapping nor fixing. Furthermore, generators are not expressed as explicit language values.

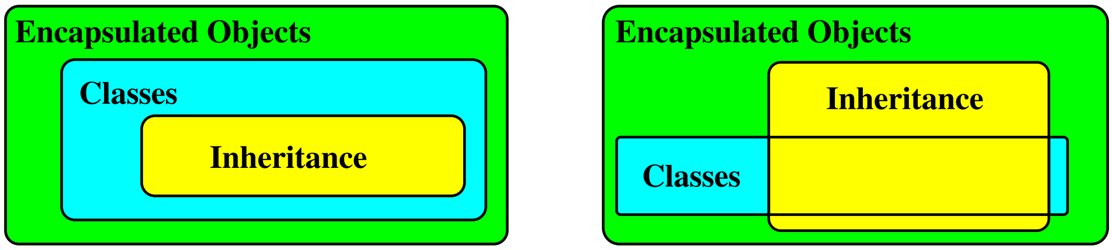
5. Discussion

5.1 Encapsulated Inheritance on Objects: A Marriage Without Unwanted Children

The most obvious difference between class-based and object-based languages is the difference in inheritance models (i.e. inheritable entities). As is widely known, class-based languages strictly separate their object and inheritance models into objects and classes. Object-based inheritance languages strive towards a unification of these models. The catch however is that *the shift from class-based inheritance to conventional object-based inheritance also involves a dramatical shift in the object model*. The object model employed by conventional object-based inheritance is fundamentally different as it suffers from inherent encapsulation problems.

Under the assumption of the immaculate client interface principle, conventional techniques only made class-based inheritance possible (figure 7 a). The contribution of our work is that we showed that the models *can* be unified without changing the object model. Figure 7 b illustrates the new situation: inheritance on encapsulated objects without the intermediate class construct.

Encapsulated inheritance on objects is a marriage between the object model employed by class-based languages and the inheritance model used in existing object-based inheritance. The semantic counterpart is that it inherits the object domain of the class-based languages of section 2.1 and the generator domain of object-based inheritance given in section 2.3.



(a) Figure 7 (b)

The above figure also indicates that this allows a possible reconsideration of reintroducing classes in prototype-based languages, for example as a classification mechanism on objects. This problem can now be tackled independently of inheritance.

5.2 Overcoming the Limitations of our Example Mechanism

One of the most fundamental criticisms against mixin-methods is that an object is required to know all its possible extensions in advance.

We claim that this criticism can be refuted by fine tuning the particular way encapsulated generators are brought back into the language development space. Although it requires more research, we already have a few (unstable) versions of Agora in which encapsulated generators can be exposed in a controlled manner.

Figure 8 shows a very simple example language construct that strives to overcome this criticism. By placing objects together in a module, a controlled exposure of specialisation interfaces is achieved. Objects in the same module can extend each other directly while objects outside the module must go through the client interface. At the semantic level this is achieved by creating a generator environment of all the objects in the module and wrapping this environment - as a background operation - in every object belonging to the module.

Encapsulation Boundary A
Object 1

Object 2
    extend object 1 **with** [...] **via** A

Figure 8

A second possible criticism is that the current version of MiniMix (and Agora) allows a mixin-method to see the implementation details of some of its parents, because of the nested scoping of mixin-methods. This is an orthogonal language design decision. Flavours of MiniMix in which the visibility of parent attributes is restricted (as in [Snyder86]) can easily be defined within the current framework. For example, a version in which a distinction is made between private variables and protected variables in the style of C++, is perfectly imaginable.

A problem related to this is making specialisation interfaces explicit (see [Lamping93]). This is currently under investigation in order to make a statically typed version of MiniMix. Important for the claim made in this paper is that this is, again, orthogonal to the inheritance mechanism underlying MiniMix.

6. Conclusion

We have shown how object-based inheritance suffers from an inherent encapsulation problem concerning specialisation interfaces. We also pointed out that this problem is absent in pure class-based languages where a strict separation between inheritable and non-inheritable entities exists. The source of the problem can be traced back to the semantic level. Although classes and objects can share a common inheritance mechanism, object-based inheritance uses a model of objects that is fundamentally different and does not allow encapsulation.

This enforces us to combine the advantages and omit the shortcomings of both. In a denotational framework, this boils down to actually making a mixture of class-based and object-based inheritance. The result - encapsulated inheritance on objects - combines the clean object model of class-based languages with the more orthogonal inheritance model of object-based inheritance.

Finally, we have proposed an example encapsulated inheritance mechanism under the form of mixin-methods which can be seen as a draft proposal for encapsulated inheritance on objects.

7. Acknowledgements.

We owe our gratitude to Theo D'Hondt, Carine Lucas, Serge Demeyer, Wim Codenie, Tom Mens, Marc Van Limberghen, Kim Mens, Niels Boyen and Kris De Volder for reading and giving comments on earlier draft versions of this paper. We also thank Theo D'Hondt for promoting this work.

References.

[Bracha92]
G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. *PhD thesis, Department of Computer Science, University of Utah, March 1992.*

[Bracha&Cook90]
G. Bracha and W. Cook. Mixin-based Inheritance. *In Proceedings of the ACM Joint OOPSLA/ECOOP'90 Conference, ACM Sigplan Notices, 25(10), pp.303-311, ACM Press 1990.*

[Codenie&al.94]
W. Codenie, K. De Hondt, T. D'Hondt, P. Steyaert. Agora: Message Passing as a Foundation for Exploring OO Languages. *To appear in SIGPLAN Notices.*

[Cook&Palsberg89]
W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. *In Proceedings of the OOPSLA'89 Conference, ACM Sigplan Notices Vol. 24(10), pp433-443, ACM Press 1989.*

[Dony&al.92]
C. Dony, J. Malenfant, P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *In Proceedings of the OOPSLA'92 Conference, ACM Sigplan Notices Vol. 27(10), pp201-217, ACM Press 1992.*

[Hense92]
A.V. Hense. Denotational Semantics of an Object-oriented Programming Language with Explicit Wrappers. *Formal Aspects of Computing, pp. 3:1-000, 1992.*

[Kiczales&Lamping92]
G. Kiczales and J. Lamping. Issues in the Design and Documentation of Class Libraries. *In Proceedings of the OOPSLA'92 Conference, ACM Sigplan Notices Vol. 27(10), pp435-451, ACM Press 1992.*

[Lamping93]
J. Lamping. Typing the Specialisation Interface. *In Proceedings of the OOPSLA'93 Conference, ACM Sigplan Notices Vol. 28(10), pp201-214, ACM Press 1993.*

[Lieberman86]
H. Lieberman. Using Prototypical Objects to Implement Shared Behaviour in an Object-Oriented System. *In Proceedings of the OOPSLA'86 Conference, ACM Sigplan Notices, 21(11), pp. 214-223, ACM Press, 1986.*

[Lucas&Steyaert94]
C. Lucas, P. Steyaert. Modular Inheritance of Objects Through Mixin-Methods. *In Proceedings of JMLC, (Peter Schulthess, ed.), pp. 273-282, Universitätsverlag ulm GmbH, 1994.*

[Mens&al.94]
T. Mens, K. Mens, P. Steyaert. OPUS, a Formal Approach to Object-Orientation. *In Proceedings of FME'94, Formal Methods Europe, pp. 326-345, Lecture Notes in Computer Science 873 (M. Naftalin, T. Denvir, M. Bertran EDS.), Springer-Verlag, 1994.*

[Nierstrasz92]
O. NierStrasz. A Survey of Object-Oriented Concepts. *In Object-Oriented Concepts, Databases, and Applications, W. Kim and F.H. Lochovsky (Eds.), pp. 3-21, ACM Press and Addison-Wesley, 1989.*

[Schmidt88]
D.A. Schmidt. Denotational Semantics: A Methodology for Language Development. *Wm C. Brown Publishers (Dubuque Iowa), 1988.*

[Snyder86]
A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *In Proceedings of the OOPSLA'86 Conference, ACM Sigplan Notices, 21(11), pp 38-45, ACM Press, 1986.*

[Stein87]
L.A. Stein. Delegation is Inheritance, *In Proceedings of the OOPSLA'87 Conference, ACM Sigplan Notices, 22(12), pp. 138-146, ACM Press, 1987.*

[Stein&al.89]
L..A. Stein, H. Lieberman, D. Ungar. A Shared View of Sharing: The Treaty of Orlando. *In Object-Oriented Concepts, Databases, and Applications, W. Kim and F.H. Lochovsky (Eds.), pp. 31-48, ACM Press 1989.*

[Steyaert94]
P. Steyaert. Open Design of Object-Oriented Languages: A foundation for Specialisable Reflective Language Frameworks. *Phd. thesis Vrije Universiteit Brussel, 1994.*

[Steyaert&.al..93]
P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen. Nested Mixin-Methods in Agora. *In Proceedings of ECOOP '93 7th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 707, O. Nierstrasz (Ed.), pp. 197-219, Springer-Verlag, 1993.*

[Wegner87]
P. Wegner. Dimensions of Object-Based Language Design. *In Proceedings of the OOPSLA'87 Conference, ACM Sigplan Notices, 22(12), pp.168-182, ACM Press, 1987.*

[Wegner&Zdonik88]
P. Wegner, S. B. Zdonik. Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like. *In Proceedings of ECOOP'88 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 322, S. Gjessing and K. Nygaard (Eds.), pp.55-77, Springer-Verlag 1988.*

A Smalltalk-implementation of Agora and a few Scheme-implementations of MiniMix are FTP-able from `progftp.vub.ac.be`. See also `progwww.vub.ac.be/prog/home.html`.

Appendix : The Definition of MiniMix

Syntax

I $\in$ **Ident** , OE $\in$ **ObjExpr**, B $\in$ **Basic** , ME $\in$ **Msg**, P $\in$ **Program**, AB $\in$ **Abstr**, AD $\in$ **AttDec**

P $\rightarrow$ OE

OE $\rightarrow$ ME | I | B | ”**self**” | ”**super**” | ”**object**” AB

ME $\rightarrow$ ”**send**” OE_1 I ”(” OE_2 ”)”

AB $\rightarrow$ ”[” AD ”]”

AD $\rightarrow$ ”**method**” I_1 ”(” I_2 ”)” OE | ”**mixin**” I_1 ”(” I_2 ”)” AB | AD_1 ”**;**” AD_2

Semantic domains

Object	= Ident $\rightarrow$ Attribute
Attribute	= Object $\rightarrow$ Object
Generator	= Generator $\rightarrow$ Object
Wrapper	= Object $\rightarrow$ Generator
Env	= Ident $\rightarrow$ Object

Semantic Functions

$\mathbb{P}$: **Program**		$\rightarrow$ Object
$\mathbb{B}$: **Basic**		$\rightarrow$ Object
$\mathbb{OE}$: **ObjExpr**	$\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator	$\rightarrow$ Object
$\mathbb{ME}$: **Msg**	$\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator	$\rightarrow$ Object
$\mathbb{AB}$: **Abstr**	$\rightarrow$ Env	$\rightarrow$ Wrapper
$\mathbb{AD}$: **AttDec**	$\rightarrow$ Env $\rightarrow$ Object $\rightarrow$ Generator	$\rightarrow$ Object

Semantic clauses

$\mathbb{P}\llbracket \text{OE} \rrbracket = \text{Wrap}(\mathbb{OE}\llbracket \text{OE} \rrbracket \perp \perp)$

$\mathbb{OE}\llbracket \text{ME} \rrbracket = \lambda e.\lambda p.\lambda g.\mathbb{ME}\llbracket \text{ME} \rrbracket\ e\ p\ g$

$\mathbb{OE}\llbracket \text{I} \rrbracket = \lambda e.\lambda p.\lambda g.(e\ \text{I})$

$\mathbb{OE}\llbracket \text{B} \rrbracket = \lambda e.\lambda p.\lambda g.\mathbb{B}\llbracket \text{B} \rrbracket$

$\mathbb{OE}\llbracket \textbf{self} \rrbracket = \lambda e.\lambda p.\lambda g.(\text{Wrap}\ g)$

$\mathbb{OE}\llbracket \textbf{super} \rrbracket = \lambda e.\lambda p.\lambda g.p$

$\mathbb{OE}\llbracket \textbf{object}\ \text{AB} \rrbracket = \lambda e.\lambda p.\lambda g.(\text{Wrap}\ (\mathbb{AB}\llbracket \text{AB} \rrbracket\ e \perp \perp))$

$\mathbb{ME}\llbracket \textbf{send}\ \text{OE}_1\ \text{I}\ (\text{OE}_2) \rrbracket = \lambda e.\lambda p.\lambda s.((\mathbb{OE}\llbracket \text{OE}_1 \rrbracket\ e\ p\ g)\ \text{I})\ (\mathbb{OE}\llbracket \text{OE}_2 \rrbracket\ e\ p\ g)$

$\mathbb{AB}\llbracket [\ \text{AD}\ ] \rrbracket = \lambda e.(\mathbb{AD}\llbracket \text{AD} \rrbracket\ e)$

$\mathbb{AD}\llbracket \text{AD}_1\ \textbf{;}\ \text{AD}_2 \rrbracket = \lambda e.\lambda p.\lambda g.(\mathbb{AD}\llbracket \text{AD}_1 \rrbracket\ e\ p\ g) +_r (\mathbb{AD}\llbracket \text{AD}_2 \rrbracket\ e\ p\ g)$

$\mathbb{AD}\llbracket \textbf{method}\ \text{I}_1\ (\text{I}_2)\ \text{OE} \rrbracket = \lambda e.\lambda p.\lambda g.\big[\ \text{I}_1 \rightarrow \lambda o.\mathbb{OE}\llbracket \text{OE} \rrbracket\ e[\text{I}_2/o]\ p\ g\ \big]$

$\mathbb{AD}\llbracket \textbf{mixin}\ \text{I}_1\ (\text{I}_2)\ \text{AB} \rrbracket = \lambda e.\lambda p.\lambda g.\Big[\text{I}_1 \rightarrow \lambda o.\big(\text{Wrap}\ \big(\text{Inherit}_{\text{Objects}}\ g\ \ (\mathbb{AB}\llbracket \text{AB} \rrbracket\ e[\text{I}_2/o])\big)\big)\Big]$