Vrije Universiteit Brussel
Faculteit Wetenschappen

Modular Inheritance of Objects Through Mixin-Methods

Carine Lucas, Patrick Steyaert

Techreport vub-prog-tr-94-07

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

Fax: (+32) 2-641-3495
Tel: (+32) 2-641-3308
Anon. Ftp: progftp.vub.ac.be

Modular Inheritance of Objects Through Mixin-Methods

Carine Lucas, Patrick Steyaert
Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels BELGIUM
email: {clucas | prsteyae}@vnet3.vub.ac.be

Abstract

In object-oriented programming languages the class concept is heavily overworked. To simplify things, there is a tendency to unbundle the different roles they play by trying to create more modular inheritance operators [Bracha&Lindstrom92]. This offers the advantage that classes can be composed in a modular way. Prototype-based languages on the other hand, also provide a simpler view on object-oriented programming, but there modular composition is totally neglected.

We introduce a generalisation of mixin-based inheritance that combines the advantages of both classes and prototypes. Mixin-methods introduce modular composition in prototype-based languages.

Keyword Codes: D.1.5, D.3.2, D.3.3

Keywords: Object-oriented Programming, Language Classifications, Language Constructs and Features

1. Modularity in Object-Oriented Languages

When discussing modularity in object-oriented languages at least two different tracks can be followed. On the one hand efforts have been made to let modules (similar to those in e.g. Modula-2) and classes/objects coexist in one language. Modula-3 and Oberon are examples of such languages. In that view modularity is regarded from an encapsulation perspective.

The other possibility is to look at a modular way of composing objects and/or classes. We refer e.g. to the Jigsaw language [Bracha&Lindstrom92], where modules are the primary definitional construct and concepts as multiple inheritance, mixins, encapsulation and strong typing are all modelled through a set of module operators. Here modularity is regarded from a composition perspective.

It is on this second track that this paper should be situated. Nevertheless, we want to start out with a short discussion of the concepts used in both approaches.

1.1 Modularity and Encapsulation

Conventionally modules are used as a name space management and encapsulation mechanism. A module defines an interface — a set of public definitions — and encapsulates the implementation of this interface. The essence of *module-based encapsulation* is that an item in a module's interface can access the implementations of all the module's items. In other words, with modules the implementation details are mutually visible to all items of the module. *A language is called modular if it has language constructs that support encapsulation and the definition of interfaces.*

Object-oriented programming languages share their concern about interfaces and encapsulation with modules.With *object-based encapsulation* a method of some object only has access to the encapsulated part of one object: the receiver. With module-based encapsulation all items that share the same interface can access each others implementation details, whereas for object-based inheritance this is not the case [Cook90].

Some object-oriented languages employ a form of *class-based encapsulation*. With class-based encapsulation attributes can be declared private, but in contrast with object-based encapsulation, all objects of the same class can invoke or access each others private attributes. Class-based encapsulation is more akin to the more general form of module-based encapsulation.

Existing languages mostly employ object-based encapsulation. In its ideal form module-based encapsulation should be an extra form of encapsulation on top of this, although this is not always the case (e.g. C++ has no object-based encapsulation). Furthermore, module-based and class-based encapsulation are often blended. It will be argued further on why this is not desirable.

1.2 Modularity and Composition

More recently the notion of modular composition has been studied. The emphasis here is on how modules can be combined to form new ones. This is done by a set of composition operators. *In this approach a language is called modular if it supports modular composition.*

In object-oriented languages modular composition is investigated as an alternative for inheritance[Bracha&Lindstrom92]. The idea of inheritance is that a new class is defined by how it differs from an already existing class. This process is referred to as *incremental modification* [Wegner&Zdonik88]. Consider a simplified model of inheritance as an incremental modification mechanism [Wegner&Zdonik88]. A parent P (the superclass) is transformed with a modifier M to form a result $R = P \Delta M = P + M(P)$ (the subclass); the result R can be used as a parent for further incremental modification. The modifier M is parameterised by a parent P to model that a subclass can invoke operations defined in the superclass. With conventional inheritance the modifier M has no existence of its own and generally is more or less part of the result R.

As opposed to this asymmetric way of working a more symmetric and more modular mechanism can be devised. Viewing the modifier M as an abstraction that exists apart from parent and result is the essence of this approach. In [Bracha&Lindstrom92] a suite of composition operators on modules is proposed, independently controlling effects as combination, incremental modification, encapsulation, name resolution and sharing. They claim that modularity is not only not in conflict with inheritance, but is its foundation.

2. Classes versus Prototypes

We will discuss the two most popular inheritance mechanisms in this section and propose an alternative in the next one. Two criteria will be used: modularity as defined above and orthogonality of language design. We use orthogonality as a criterion for the quality of a programming language, thus claiming that separate concepts have to be introduced to model separate functionalities.

2.1 Why Class-Based Languages Are Not Modular

It is clear that current class-based programming languages do not satisfy our definition of modularity as there is no modular composition. It is however possible to achieve modular composition in a class-based system, by introducing stand alone modifiers [Bracha & Cook90].

However, the principle of orthogonality is severly violated by the class concept. Among others, classes are used for classification, incremental modification, constructing instances, determining attribute visibility and typing. [Bracha&Lindstrom92] enumerates no less than 11 different roles classes play. It is widely understood that several concepts have been confused with the notion of a class. Witness of this are numerous proposals to differentiate classes from types [Canning et al.89], from modules [Szyperski92] and to provide sharing mechanisms that are independent of the class construct.

We therefore agree with [Szyperski92] that some sort of module-based encapsulation is indeed very useful in some cases, but that classes and modules are separate concepts. Therefore module-based encapsulation should not be strictly coupled to classes, but rather modules should be provided as an explicit language construction.

2.2 Why Prototype-Based Languages Are Not Modular

The unbundling of the different roles played by classes is one way to try to simplify things. Parallel to this evolution is the development of prototype-based languages. They also provide a simpler view on object-oriented programming. However, in prototype-based languages modularity is often neglected.

Classless languages employ a delegation mechanism rather than inheritance. Delegation is a message forwarding mechanism for objects. With *explicit delegation* an object can explicitly delegate a message to any object it has knowledge of. With *implicit delegation* an object can designate another object as its parent to which messages will be delegated. Once again, two design choices can be made depending on whether the delegation structure (i.e. the parent) can be dynamically changed or not. [Stein,Lieberman &Ungar89] calls the former *unanticipated delegation*, the latter is called *anticipated delegation.*

The objection we make to both explicit delegation and implicit unanticipated delegation is that they can not be recast in terms of modular composition [Steyaert94].

Both implicit anticipated delegation and classes are however in essence based on an incremental modification mechanism. Whereas in class-based languages inheritance involves incremental modification of classes, implicit anticipated delegation can be considered as incremental modification of objects. It is therefore possible to achieve modular composition in a system with implicit anticipated delegation also. Rather than composing classes with stand alone modifiers, objects will be composed this way. We will propose a generalisation of mixin-based inheritance that is such a mechanism.

3. Mixin-Based Inheritance and Mixin-Methods

After discussing conventional mixin-classes and mixin-based inheritance, we will introduce mixin-methods and show in what way they are modular.

3.1 Mixin-Classes

In multiple inheritance languages that linearise the inheritance graph (e.g. CLOS [Moon89]), it is possible to have classes that have no apparent ancestor but that do invoke parent operations in a meaningful way. This sort of classes has to rely on linearisation to be ‘mixed in’ at the appropriate place in the linearised inheritance hierarchy (i.e. as inheritor from a class that provides the necessary operations). These classes have therefore been named *mixin-classes*. The effect is that it is possible to create mixin-classes that can be applied to a set of different superclasses.

The prototypical example is that of a colour mixin, that adds a colour attribute and the associated access methods, and can be applied to classes as different as vehicles and polygons. A typical example involving the invocation of parent operations is the “bounds” mixin that puts boundaries on the co-ordinates of a geometric figure. The actual base class can be taken from a set of possible classes. This could be, amongst others, a class Point, a class Line or a class Circle.

3.2 Mixin-Based Inheritance

Contrary to mixin-classes, in *mixin-based inheritance*, a mixin is not a class (a mixin cannot be instantiated for example), and multiple inheritance is a consequence of, rather than the supporting mechanism for, the use of mixins. In contrast to CLOS, in which mixins are nothing but a special use of multiple inheritance, mixins are promoted as the only abstraction mechanism for building the inheritance hierarchy [Bracha&Cook90] [Bracha92] [Hense92] [Steyaert & al.93].

Referring back to our model of inheritance as an incremental modification mechanism, the essence of mixin-based inheritance is exactly to view the modifier M as an abstraction that exists apart from parent and result. Modifiers are called mixins. The composition operation Δ is called mixin application. The class to which a mixin is applied is called the base class. In pure mixin-based inheritance, classes can only be extended through application of mixins (see Listing 1).

The Δ operator sees to it that the parent P is passed as explicit parameter to the modifier M. In practice a mixin does not have its base class as explicit parameter, but rather, a mixin has access to the base class through a pseudo variable, in the same way that a subclass has access to a superclass through a pseudo variable (e.g. the `super` variable in Smalltalk). In a statically typed language, though, this means that a mixin must specify the names and associated types of the attributes a possible base class must provide. This is why mixins are sometimes called abstract subclasses.

```
class-based inheritance
class R1
 inherits P1
 extended with NamedAttribute1 ... NamedAttributen
endclass

class R2
 inherits P2
 extended with NamedAttribute1 ... NamedAttributen
endclass

mixin-based inheritance
M is mixin
  defining NamedAttribute1 ... NamedAttributen
  applicable to base class with[1]
    SuperAttributeSignature1 ... SuperAttributeSignaturem
  endmixin
class R1 is P1 extended with M endclass
class R2 is P2 extended with M endclass
```

Listing 1: Class-based versus mixin-based inheritance

3.3 Mixin-Methods

We suggest an approach, as used in the Agora language, that differs from classical mixin-based inheritance. For reasons of simplicity though, we will use an informal object-oriented syntax here. For an extensive discussion of the Agora language we refer to [Codenie et al.94],[Steyaert94],[Steyaert et al 93].

---

[1] For reasons of brevity, this specification will be omitted in further examples.

Applying the orthogonality principle to the facts that we have mixins and that an object consists of a collection of named attributes, one must address the question of how a mixin can be seen as a named attribute of an object. The adopted solution is that an object lists as mixin attributes all mixins that are applicable to it. The mixins that are listed as attributes in a certain object can only be used to create extensions of that object and its future extensions. Furthermore, an object can only be extended by selecting one of its mixin attributes. In much the same way that selecting a method attribute from a certain object has the effect of executing the selected method-body in the context of that object, selecting a mixin attribute of a certain object has the effect of extending that object with the attributes defined in the selected mixin. Therefore, the terminology: *mixin-methods*. So, rather than having an explicit operation to apply an arbitrary mixin to an arbitrary object, an object is asked to extend itself.

Mixin-methods use object-based inheritance. There are no classes and new objects can only be created by copying or cloning or extending existing objects. This makes sharing of state possible, next to sharing of behaviour. Objects can be extended by sending mixins to them[2]. This can also be done dynamically, which means that it is not necessary to declare all prototypes that will be used in some program beforehand. This also means that there is no distinction between different kinds of objects, all objects are candidates for extension.

```
object SecondObject is FirstObject aMixin
```

Inheritance of mixins plays an important role in this approach. If it were not for the possibility to inherit mixins, the above restriction on the applicability of mixins would result in a rather static inheritance hierarchy and in duplication of mixin code (each mixin would be applicable to only one object). A mixin can be made applicable to more or less objects according to its position in the inheritance tree. The higher a mixin is defined the more objects that can be extended with it. In a programming language such as Agora, where mixin-based inheritance is the only inheritance mechanism available, this means that all generally applicable mixins (such as a mixin that adds colour attributes) must be defined in some given root object (that all other objects will be extensions of).

```
--- Root object attributes ---
 MakeColoured is mixin
   defining colour
 endmixin

 MakeCar is mixin
   defining enginetype
 endmixin

object Car is RootObject MakeCar
 --- object Car inherits MakeColoured defined on the Root object
object ColouredCar is Car MakeColoured
```

Listing 2: Inheritance of mixin methods

3.4 Why Is This Modular ?

Mixins provide the extra abstraction that is needed to be able to construct an entire inheritance hierarchy in a modular fashion. Furthermore, extension of objects is obtained in a very object-oriented way.

Consider the multiple inheritance hierarchy in Figure 1. This hierarchy models a variety of classes. The root class `BasicClass` implements the standard behaviour of a class as a template. Other functionalities that can be assigned to classes are: they can be inheritable, they

---

[2] We therefore use a Smalltalk-like message passing syntax.

can have a name, they can hold a collection of all their instances, ...[3] Furthermore, combinations of these functionalities are possible with the result that we get a complicated, extremely tangled hierarchy.

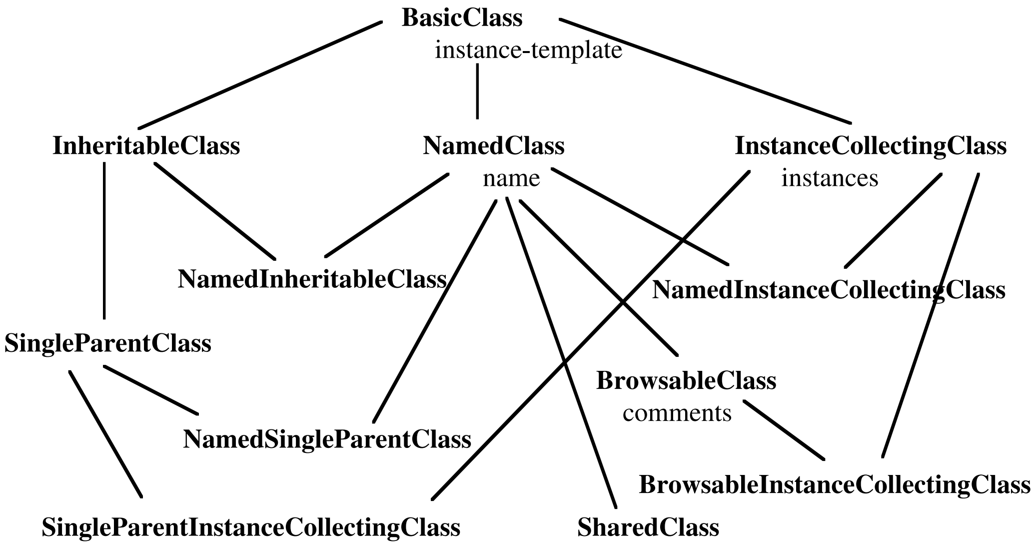

Figure1: A multiple inheritance hierarchy modelling classes

If we try to model this same hierarchy with mixins, we get the situation as sketched in Figure 2. Every different functionality is modelled by a different mixin and to create a certain kind of class one just needs to combine the right mixin-methods.

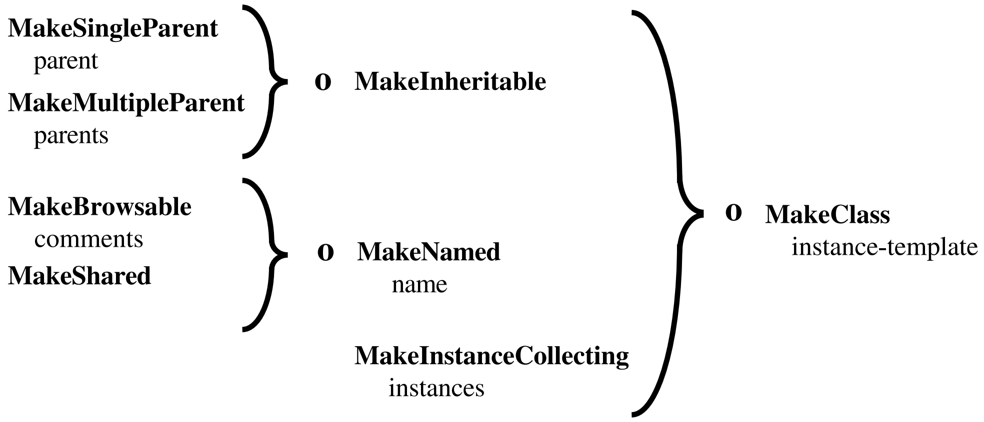

Figure 2: Classes modelled by mixin-methods

We mentioned above that every object responds to a limited set of mixin messages. This is expressed by the structure of Figure 2. The mixin `MakeClass` will typically be defined on the root object. The mixins `MakeInheritable`, `MakeNamed` and `MakeInstanceCollecting` are defined in the mixin `MakeClass`. This means that only

[3] Note how this example demonstrates once more how overworked the class concept is.

objects to which the mixin `MakeClass` has already been applied understand these three mixins. In other words, these mixins can only be applied *after* (the symbol **o**) `MakeClass`. The same holds for the mixins on the left.[4]

Listing 3 clearly demonstrates how e.g. the mixins `MakeInheritable`, `MakeNamed` and `MakeInstanceCollecting` are defined in the mixin `MakeClass`. It furthermore demonstrates how objects are created in a highly modular fashion. First the object `aClass` is created by sending the mixin `MakeClass` to the `RootObject`. By doing this the mixins `MakeInheritable`, `MakeNamed` and `MakeInstance-Collecting` are defined on the object `aClass`, so that it is now possible to send `MakeNamed` to `aClass`. The same goes for the mixin `MakeBrowsable`.

```
...
MakeClass is mixin
  defining
    instance-template is variable
    MakeInheritable is mixin
      defining ...
      endmixin
    MakeNamed is mixin
      defining
        name is variable
        MakeBrowsable is mixin
          defining
            comments is variable
          endmixin
        ...
      endmixin
    MakeInstanceCollecting is mixin
      defining
        instances is variable
      endmixin
  endmixin

object aClass is RootObject MakeClass
object aNamedClass is aClass MakeNamed
object FirstBrowsableClass is aNamedClass MakeBrowsable
object SecondBrowsableClass is aNamedClass MakeBrowsable
```

Listing 3: The class example

Besides creating the ability to construct inheritance hierarchies in a modular fashion, our system is also modular in its own conception. We discussed above how e.g. the class concept is heavily overworked and that we follow the orthogonality principle in language design. In the core of our system objects model object-based encapsulation and modular composition models inheritance. Other concepts can be added to this core in an orthogonal way. Classes could be added as a classification mechanism (and nothing else !!), modules could be added to achieve module-based encapsulation and thus separate compilation, explicit types could be added for typing purposes, etc. Referring to the general belief that OO = objects + classes + inheritance,

---

[4] Note that the set of mixin-methods in Figure 2 allows you to create even more classes than are shown in Figure 1. We did however not draw all possible combinations in Figure 1 to avoid overloading the figure.

we demonstrated the alternative OO = objects + composition *(+ classes) (+modules) (+types) (+…).*

4. The Power of Mixin-Methods

We would like to conclude with some illustrations of the power and expressivity of mixin-methods.

4.1 Dynamic Mixin Application

As mentioned earlier on, extension of objects through mixin application can also be achieved dynamically. This means that if we return to our code fragment from above, we get the following situation. We first made an object `aClass`, that we extended to get `aNamedClass`, by sending the mixin `MakeNamed` to it. Now imagine that two users use this same class, but both want to add their own comments. This is achieved by making two objects `FirstBrowsableClass` and `SecondBrowsableClass`, that share the 'MakeNamed-part'. When a message is sent to `FirstBrowsableClass` that e.g. changes the name of this class, this will also be visible to `SecondBrowsableClass` and vice versa.

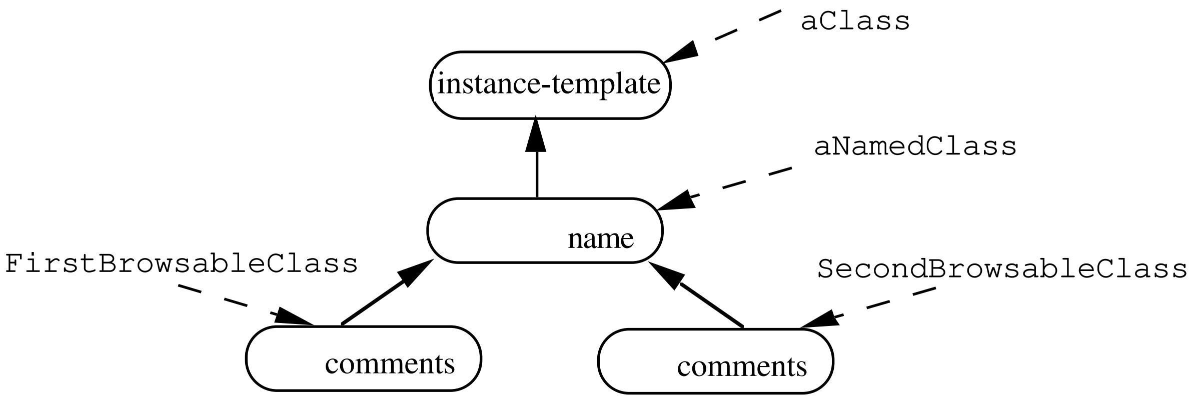

Figure 3: Dynamic Application of Mixin-Methods

4.2 Applicability of Mixin-Methods

We showed that unconstrained multiple inheritance hierarchies often end up as tangled hierarchies. Multiple inheritance is less expressive than it appears, essentially in its lack to put constraints on multiple inheritance from certain classes [Hendler86] [Hamer92]. One such constraint is a mutual exclusion constraint on subclasses. Restricting the applicability of mixin-methods puts a constraint on the possible inheritance hierarchies that can be constructed.

Consider a `Person` object with a `MakeFemale` and a `MakeMale` mixin-method (example taken from [Hamer92]). A mutual exclusion constraint on `MakeFemale` and `MakeMale` expresses that once `MakeFemale` is applied to a `Person` object, `MakeMale` should not be applicable to the resulting object, and vice versa. This mutual exclusion constraint can be realized simply by canceling `MakeMale` in `MakeFemale`, and by canceling `MakeFemale` in `MakeMale`.

A more elegant solution, and one that should be provided in a full-fledged programming language, would be to have some declarative means to express that two mixin-methods are mutually exclusive or covering or any other imaginable constraint. Classifiers [Hamer92] play this role for (non mixin-based) class-based languages. A similar mechanism is imaginable for mixin-methods.

4.3 Mixins and Object-Oriented Design

These considerations automatically lead us to object-oriented design. The example of the hierarchy of classes demonstrated clearly that each mixin used in the creation of an object can encode one specific role an object plays in its context.

In [Andersen&Reenskaug92] an object-oriented design technique based on roles and role models is introduced. They define a role model to be a unit of design, that comprises two or more interacting entities denoted as roles. Each role is considered a requirement/ responsibility of objects participating in the actual execution of the behaviour described in the role model. Roles are basically more fine-grained than classes. Typically several roles in several role models will correspond to the same real-world phenomenon to be implemented by a single class. This is exactly what we do with our mixin-methods.

Moreover we just showed that it is possible to constrain the combinations of mixins — and thus the combinations of the roles they implement. This makes an easy mapping of the design concepts to the implementation possible. A further investigation of these possibilities is one of our future goals.

4.4 Mixins and Multiple Inheritance

In [Boyen,Lucas,Steyaert94] we have shown that minor extensions to mixin-based inheritance can offer a solution to the much discussed problems involved in expressing multiple inheritance hierarchies.

5. Conclusions

Recasting inheritance in terms of modular composition is promising. Modular composition has already been studied in class-based languages. However, in class-based languages inheritance and classes are not orthogonal language concepts. Moreover they lack some of the interesting properties of prototype-based languages. In this paper we investigated mixin-methods as an inheritance mechanism for objects. We showed that mixin-methods are a form of modular composition and that mixin-method inheritance is orthogonal to the class concept. We also gave an indication of the expressive power of mixin-methods.

6. References

[Andersen&Reenskaug92] E. P. Andersen and T. Reenskaug: *System Design by Composing Structures of Interacting Objects*, In Proceedings of ECOOP '92 European Conference on Object-Oriented Programming, pp. 131-152, Springer-Verlag.

[Bracha&Lindstrom92] G. Bracha and G. Lindstrom: *Modularity meets Inheritance* In Proc. of IEEE Computer Society International Conference on Computer Languages, pp. 282-290, 1992.

[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 ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.303-311, ACM Press 1990.

[Boyen,Lucas&Steyaert94] N. Boyen, C. Lucas and P. Steyaert: *Generalised Mixin-based Inheritance to Support Multiple Inheritance* Paper joining poster session on OOPSLA ‘94. Conference on Object Oriented Programming, Systems, Languages and Applications, 1994.

[Canning et al.89] P. Canning, W. Cook, W. Hill,W. Olthoff: Interfaces for Strongly-Typed Object-Oriented Programming In Proceedings of OOPSLA‘89 Conference on Object-Oriented Programming, Systems, Languages and Applications, pp.457-467, ACM Press 1989.

[Codenie et 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 of December1994 or January 1995

[Cook90] W. Cook: *Object-Oriented Programming Versus Abstract Data Types*, In Foundations of Object-Oriented Programming Languages, Proceedings of REX School/Workshop, pp. 151-178, LNCS 489, Springer-Verlag 1990.

[Hamer92] J. Hamer: *Un-Mixing Inheritance with Classifiers* In Proceedings of ECOOP’92 Workshop on Multiple Inheritance and Multiple Subtyping, available as Working Paper WP-23 Dept. of Computer Science and Information Systems, Univ. of Jyväskylä, pp.6-9, 1992.

[Hendler86] J. Hendler: *Enhancement for Multiple Inheritance* In Proceedings of Object-Oriented Programming Workshop 86, Sigplan Notices Vol 21 (10), pp.98-106, October 1986.

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

[Moon89] D.A. Moon: The COMMON LISP Object-Oriented Programming Language Standard, Object-Oriented Concepts, Databases and Applications, Won Kim and Frederick H. Lochovsky (Eds.), pp. 79-126, ACM Press 1989.

[Stein,Lieberman&Ungar89] L.A. Stein, H. Lieberman and D. Ungar: *A Shared View of Sharing: The Treaty of Orlando* In Object-Oriented Concepts, Databases, and Applications, Won Kim, Frederick H. Lochovsky Eds, pp.31-48, ACM Press 1989.

[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 European Conference on Object-Oriented Programming, pp. 197-219, Springer-Verlag.

[Steyaert 94] P. Steyaert: *Open Design of Object-Oriented Languages A Foundation for Specialisable Reflective Language Frameworks*, PhD thesis, Vrije Universiteit Brussel, 1994

[Szyperski92] Clemens A. Szyperski: *Import is Not Inheritance Why We Need Both: Modules and Classes,* In Proceedings of ECOOP ‘92 European Conference on Object-Oriented Programming, pp. 19-32, Springer-Verlag.

[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, pp.55-77, Springer-Verlag 1988.