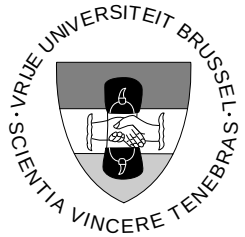
Vrije Universiteit Brussel

Faculteit Wetenschappen



Generalised Mixin-based Inheritance to Support Multiple Inheritance

Niels Boyen, Carine Lucas, Patrick Steyaert

Techreport vub-prog-tr-94-12

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

Generalised Mixin-based Inheritance to Support Multiple Inheritance

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

Abstract

The semantics of multiple inheritance is still very much a subject of debate. Before a clearer view on the issue is obtained, multiple inheritance will not enjoy wide-spread use. As stated by Alan Snyder in [Shan&al.93] "it is easy to design a multiple inheritance model that supports any given example; it is not clear how to design one that supports new examples". We argue that it *is* possible to construct a simple, comprehensible, and general multiple inheritance mechanism. This is achieved by fragmenting the functionality of multiple inheritance. For this purpose a generalised form of mixin-based inheritance is introduced. We show that these generalised mixins are exactly the right building blocks, and provide exactly the right balance between exposure and encapsulation of inheritance to express multiple inheritance hierarchies.

Keyword Codes: D.1.5, D.3.3

Keywords: Object-oriented Programming, Language Constructs and Features

1 Introduction

Whereas the semantics of single inheritance is relatively well understood, the semantics of multiple inheritance is still a debatable issue. It is not even clear whether it is possible to construct a single, simple, comprehensible, and general mechanism that solves all problems related to multiple inheritance [Shan&al.93]. We argue that this *is* possible and that the solution can be found in the fragmentation of the functionality of the inheritance mechanism into its primitive building blocks in such a way that a greater flexibility is obtained for the user to adapt the inheritance strategy to specific situations. We claim that this can be achieved by making the underlying mechanisms of inheritance explicit. In casu wrappers and wrapper application such as studied in the context of single inheritance [Cook&Palsberg89] will be used.

Making wrappers explicit results in mixin-based inheritance. We generalise mixin-based inheritance in two ways. In their original form [Moon89] [Bracha&Cook90] mixins were used to extend classes. In our case mixins are made applicable to objects to enable object-based inheritance. Second, our mixins are based on a more general form of wrappers, where wrappers can have multiple parents. This enables mixins to invoke parent operations of non-direct ancestors.

Whether name conflicts are entirely resolved in the resulting class or not is crucial to our approach. We thus distinguish two orthogonal sorts of inheritance. With the latter sort the interfaces of the combined classes are kept strictly separate, and name conflicts must be resolved when passing a message to the inheritor (e.g. by qualification). With the former kind, the interfaces of the combined classes are merged and all name conflicts are to be explicitly resolved in the inheritor. We will show that while multiple inheritance problems are manageable in both pure forms, this is less clear for hybrid mechanisms.

2 Generalised Mixin-based Inheritance

In this section we will first discuss mixin-based inheritance as it was originally introduced and then present our generalisations.

2.1 Mixin-based Inheritance

Consider inheritance as an incremental modification mechanism [Wegner&Zdonik88], where a parent P is transformed with a modifier M to form a result R = P+M(P). The modifier M is parameterised by a parent P to model the fact that a subclass can invoke operations defined in the superclass. Whereas in classical inheritance the modifier M has no existence of its own, the essence of mixin-based inheritance is exactly to view the modifier M as an abstraction that exists apart from parent and result (Listing 1). Modifiers are called 'mixins'. In 'pure' mixin-based inheritance [Bracha&Cook90], classes can only be extended through application of mixins.

In practice 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, 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'. For a more detailed study of our approach to mixins see [Steyaert et al.93], [Codenie et al.94] and [Steyaert94].

```
class-based  inheritance
class R1
 inherits P1
 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 inherits P1 extended with M endclass
class R2 inherits P2 extended with M endclass
```

Listing 1

2.2 Generalisations of Mixins

We claim that the solution to the numerous problems concerning multiple inheritance can be found in the way one looks at the inheritance graph. Consider extending a class X with two mixins AM and BM. Consider a method m that is defined on both AM and BM.

```
AM is mixin            BM is mixin
   defining               defining
      m                      m
     ...                    ...
  endmixin               endmixin
```

Listing 2

[1] This specification will be omitted in further examples for reasons of brevity.

There are two possibilities now. We either want the interfaces of AM and BM to remain separated, so that we can invoke both versions of m dependent of what view on the class we want to consider. Or we want to merge the two interfaces and need a mechanism to resolve the name conflict. We will show how these two views can be mimicked with a generalised form of mixin-based inheritance. We will therefore demonstrate how different views on objects can be installed in the first approach - by using mixins as a means for prototype-based inheritance - and introduce a mechanism to resolve name conflicts in the second - by allowing mixins to invoke parent operations of non-direct ancestors.

2.2.1 Separated Interfaces

Let us first consider the case where we want to keep the interfaces separated. In our introduction of mixin-based inheritance mixins were only used to compose classes. However, mixins can equally well be used to dynamically extend objects in a prototype-based approach to object-oriented programming. New objects can be created by taking an existing object and extending it with a set of variables and methods. Similar to mixins in a class-based language we can identify a base object and a set of extensions and extensions can be considered as separate abstractions. The terminology mixins and mixin application from the class-based case can be retained.

This approach can be used to model inheritance hierarchies, where we want to keep the interfaces of two combined classes separate. Consider wanting to extend an object x with two different mixins AM and BM.

```
x is aRootObject extended with aMixin;
...
xAsA is x extended with AM;
xAsB is x extended with BM;
```

Listing 3

In listing 3, we start with some object x. We can then create two new objects, xAsA and xAsB, each representing a different view on x. We can now send messages to xAsA and xAsB, dependent of what view on the object we want to consider. Being two dynamic extensions of x, they share its attributes (Figure 1). This means that when a message is sent to xAsA, that e.g. changes the value of some attribute in the x-part of this object, this is visible to xAsB and vice-versa.

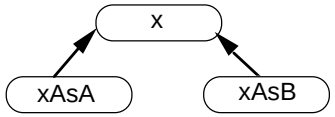


Figure 1

The drawbacks and advantages of this approach will be discussed further on. Let us first take a look at the opposite view.

2.2.2 Merged Interfaces

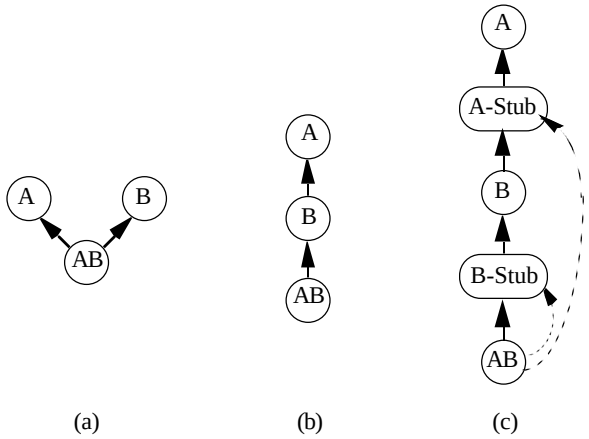
The other possibility is that we want to merge both interfaces, so that only one version of the message m is applicable to the extended objects. Using mixins in a convential way automatically generates a linearised inheritance chain, so that only one version of each message is visible. However this is not sufficient. In some cases we do not just want to take the last version of some method, but we want to combine the conflicting methods into one method.

```
ABM is mixin
   needs A-Stub B-Stub
   defining
       methods
         m
            m super:A-Stub
            m super:B-Stub
   endmixin

anAB is aRootobject
   extended with AM defining A-Stub for ABM
   extended with BM defining B-Stub for ABM
   extended with ABM
endis
```

Listing 4

We therefore create a new mixin ABM, where we override m, to invoke the m-methods of both parents (listing 4). This is achieved by using stubs.



(a) (b) (c)

Figure 2

Simple qualified message passing does not work here since a mixin does not have a single base class that could serve as a qualifier. The problem here is that we have to deal with a linearised inheritance chain (figure 2.b), but we still want to be able to refer to non-direct super classes (i.e. we want to simulate figure 2.a). To do this we have to bring some 'hierarchy' into the chain. We therefore introduced the notion of stubs to create access to non-direct parents. Just as mixins, stubs have to be inserted at the right place in the inheritance chain (figure 2.c). In this manner subclasses can use non-direct superclasses as parameters and 'mimic' a graph structure in the linear chain (dashed arrows in figure 2.c). Stubs then serve as pointers to the place in the inheritance chain where method lookup should start when invoking parent operations.

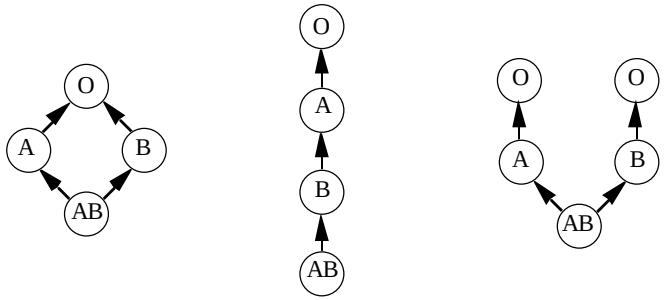
3 Need for a More "Global" View on the Inheritance Graph

3.1 The Diamond Problem

The basic problem one has to face when working directly with the multiple inheritance graph is the diamond problem depicted in figure 3.a. An inheritor AB multiple inherits from classes/objects A and B, which in turn inherit from a common ancestor O. The problems that are related to the diamond problem are:

- Encapsulation of inheritance
  Is it allowed for a client or inheritor of a class to depend on the inheritance structure of that class?
- Common ancestor duplication problem
  Must the common ancestor be duplicated or not?
- Common ancestor name conflicts problem
  Are the name collisions that arise when attributes of a common ancestor are inherited along different paths considered conflicts or not? How are these conflicts resolved?
- Duplicate parent operation invocation problem
  Multiple inheriting from two classes that both invoke the same parent operation on a common ancestor can cause duplicate invocation of this parent operation. How is this problem dealt with?

Different multiple inheritance strategies can be classified according to how they treat the inheritance graph in the presence of diamonds (figure 3). While graph multiple inheritance treats the graph as is, with linearised and tree multiple inheritance the graph is transformed.



a. Graph MI **b. Linearised MI** **c. Tree MI**

Figure 3

3.2 Need for a More "Global" View on the Inheritance Graph

Each of the above inheritance mechanisms gives different answers to the questions that the diamond problem raises. None of them gives a satisfactory solution to multiple inheritance [Snyder87][Knudsen88], because the encapsulation of inheritance is in conflict with some of the answers to the three other questions.

Graph multiple inheritance breaches the encapsulation of inheritance, because neither A nor B can decide one-sidedly *not* to inherit from O. For example, an attribute x defined on O is, according to the rules of graph multiple inheritance, not conflicting in AB. However if A is changed so that it is implementing attribute x itself, this operation becomes a conflict in AB.

Tree multiple inheritance solves the encapsulation problems by always duplicating common ancestors, but this goes at the cost of expressivity. Duplicating common ancestors is especially not advisable in inheritance hierarchies that have a single root class. Every class in the system that multiple inherits from two or more ancestors will need to resolve the conflicts that arise from the duplication of this root class. This imposes an absurd overhead on the use of multiple inheritance.

Linearisation inheritance breaches encapsulation since an inheritor cannot reliably communicate with its direct ancestors. Due to linearisation sometimes unrelated classes/objects are inserted between an inheritor and one of its direct ancestors. This is the case for B in the linearised inheritance graph above. Even though in the graph representation B has O as direct ancestor, in the linearised graph A has become the direct ancestor of B (O becomes an indirect ancestor of B). Especially when name conflicts are involved this can give surprising results.

So, there is a trade-off between expressivity and encapsulation. Encapsulation of inheritance can be interpreted as imposing a localised view on the inheritance graph for each class. In some cases a "global" view on the inheritance graph is needed to achieve the necessary expressivity. We thus conclude that it is sometimes necessary to expose the inheritance structure in a controlled way.

4 A Global View on the Inheritance Graph with Mixins

We claim that our generalised mixin-based inheritance provides exactly the right balance between encapsulation of inheritance and expressivity. Since a mixin is an abstract subclass, the parent operations it invokes are part of its interface. Furthermore, mixin-based inheritance causes explicitly linearised inheritance. Attributes in the mixin override the attributes of the base class/object having the same name. In the absence of any name clash resolution mechanism, attribute name lookup is determined by application order.

Let us first take a look at the common ancestors name conflicts problem. The first thing to do when constructing a branch of the inheritance hierarchy using multiple inheritance is to decide whether one wants to merge or separate the interfaces of the combined classes. When considering merging the interfaces, we can again distinguish two cases. In the first case we want to combine the methods of the combined objects in a new method on the new object. This case was handled in section 2.2.2, where we introduced stubs. Using these stubs, a mixin ABM was created, that solves the name conflicts appearing when A and B are combined. We should note that to avoid problems with self references, *all* name conflicts have to be explicitly resolved here. It is not sufficient to simply resolve name conflicts occurring through combination of A and B. It is equally possible that self sends are performed of methods that are implemented in one of the ancestors of A or B. It is therefore necessary to resolve *all* name conflicts in ABM. Furthermore, the use of stubs must be restricted so that they can only be used to invoke parent operations of non-direct parents and so that they cannot put constraints on the order in which mixins are applied.

The second possibility concerning the merging of interfaces is that A and B both define m by invoking a super-call of the method m. Working directly on the multiple inheritance graph would in this case not only cause problems with name conflicts, but would also cause each message m sent to anOAB to result in two messages m to the O subobject.


```
OM is mixin
   defining
     m
       ...
  endmixin

AM is mixin
   defining
     m
       ...
        super m
  endmixin

BM is mixin
    defining
     m
       ...
        super m
  endmixin

anOAB is aRootobject
  extended with OM
  extended with AM
  extended with BM
endis
```

Listing 5

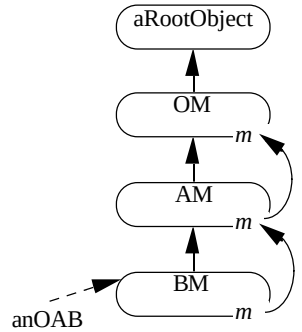


Figure 4

In an encoding with linearised multiple inheritance the anOAB method m *can* only invoke the method m of its ancestor once (listing 5, figure 4). Furthermore, as the programmer has total control over the linearisation, there are no unforeseen insertions of unrelated classes between a class and its parent. This makes parent invocations safe. Note that the above solution is heavily based on global information of the inheritance graph. The example can only be constructed as a linear chain since we have information about the way each of the subobjects invokes parent operations.

Our second approach consists of keeping the interfaces of the combined classes separated and offering a way to express which method m is wanted. In section 2.2.1. we introduced separated interfaces through a prototype-based approach to mixin-based inheritance. If we now send the message to e.g. xAsA, this may lead to the evaluation of e.g. '**self** m (Figure 5). In our approach '**self** m' is accordingly sent to this same initial receiver object (xAsA). All 'naive' approaches to qualified message passing will fail to correctly interpret this sort of programs.

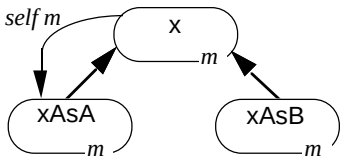


Figure 5

Furthermore, qualified message passing can disable further refinement of certain attributes, by encoding information about the class hierarchy as constant information [Bobrow et. al.86]. When e.g. all messages y to self, within the implementation of a class A are qualified with class name A, then inheritors of class A can not refine (overwrite) the method y (genericity inhibition problem in [Carré&Geib90]). This problem is also resolved in our approach.

Regarding the common ancestor duplication problem and duplicate parent operation invocation problem we should remark that this can be controlled explicitly by the programmer as well: not by the order of application, but by the number of applications of one and the same mixin.

5 Open Questions

The approach suggested in this paper was based on two different views on the inheritance hierarchy. In one view the interfaces of the combined classes were merged, in the other they were kept separate. One question now suggests itself: should it be possible to combine these two views? In other words, should we be able to merge and separate interfaces within one single branch of the hierarchy? Another question concerns the distinction we make to solve name collisions in merged interfaces. In one approach we use stubs, in the other the collisions are automatically resolved through linearisation. Will this distinction always be clear? The solution to these problems will probably have to be sought in a good design of mixins.

A third problem concerns the exact semantics of these mixins. We now have several entry-points to the object `x` (`x`, `xAsA`, `xAsB`), where we can all send messages to. Certain problems concerning self sends may still occur. These problems are related to the problems with split objects [Dony et al 92] and to the modelling of inheritance with explicit bindings [Hauck93]. Reaching a full understanding of these problems and finding an adequate solution are the future goals of our research.

6 Conclusions

We confirm with [Knudsen88] that: "*... by choosing strict and simple inheritance rules, one is excluding some particular usage of multiple inheritance ...*". We add to this conclusion that there is a trade-off between full encapsulation of inheritance and the expressivity of the inheritance strategy and, consequently, that it is sometimes necessary to expose the inheritance structure in a controlled way. This is e.g. apparent in the case where we use linearisation to solve name collisions.

A possible solution could be devised where the programming language provides different inheritance operators. This is more or less the direction taken in [Knudsen88]. We proposed a different solution based on mixins. We generalised mixin-based inheritance in two ways. First, mixins were made applicable to objects to enable object-based inheritance. Second, we made parent bindings explicit. We then showed that, given these generalisations, mixins are sufficient to express all the above multiple inheritance hierarchies in an effective and simple way.

References

[Bobrow et al 86] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel: *CommonLoops Merging Lisp and Object-Oriented Programming*, In Proceedings of OOPSLA'86, pp17-29.

[Bracha&Cook90] G. Bracha and W. Cook: *Mixin-based Inheritance* In Proc. of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.303-311, ACM Press 1990.

[Carré&Geib90] B. Carré and J. Geib: *The Point of View notion for Multiple Inheritance* In Proc. of ACM Joint OOPSLA/ECOOP'90 Conference Proceedings, pp.3312-321, ACM Press 1990.

[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 December 1994 or January 1995

[Cook&Palsberg89] W. Cook and J. Palsberg: *A denotational semantics of Inheritance and its Correctness*, In Proceedings of OOPSLA'89, pp. 433-443.

[Dony et al 92] C. Dony, J. Malenfant and P. Cointe: *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation* In Proceedings of OOPSLA '92, pp. 201-217.

[Hauck93] F. Hauck: *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance* In Proceedings of OOPSLA '93, pp. 231-239

[Knudsen88] J. Lindskov Knudsen: *Name Collision in Multiple Classification Hierarchies*, In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pp. 93-109, Springer-Verlag 1988.

[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.

[Shan&al.93] Y. Shan, T. Cargil, B.Cox, W. Cook, M. Loomis, A. Snyder: *Is Multiple Inheritance Essential to OOP*, In Proc. of OOPSLA'93, pp. 360-363, September 1993.

[Snyder87] A. Snyder: *Inheritance and the Development of Encapsulated Software Components*, In Research Directions in Object-Oriented Programming, B. Shriver and P. Wegner (eds), pp 165-188, MIT Press 1987.

[Steyaert et al.93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, M. Van Limberghen: *Nested Mixin-Methods in Agora*, ECOOP '93 European Conference on Object-Oriented Programming, pp. 197-219, Springer-Verlag .

[Steyaert94] P. Steyaert: *Compositionality and Abstraction in Agora, A Framework for Reflective Object Oriented Programming Languages*, PhD thesis, Department of Computer Science, Brussels Free University, 1994.

[Wegner&Zdonik88] P. Wegner, S. B. Zdonik: *Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like*. In Proc. of ECOOP'88 European Conference on Object-Oriented Programming, pp.55-77, Springer-Verlag 1988.