Nested Mixin-Methods in Agora

Patrick Steyaert*, Wim Codenie, Theo D'Hondt,
Koen De Hondt, Carine Lucas, Marc Van Limberghen

Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels BELGIUM
email: {prsteyae | wcodenie | tjdhondt | kdehondt |
clucas | mvlimber}@vnet3.vub.ac.be

Abstract: *Mixin-based inheritance is an inheritance technique that has been shown to subsume a variety of different inheritance mechanisms. It is based directly upon an incremental modification model of inheritance. This paper addresses the question of how mixins can be seen as named attributes of classes the same way that objects, methods, and also classes in their own right, are seen as named attributes of classes. The general idea is to let a class itself have control over how it is extended. This results in a powerful abstraction mechanism to control the construction of inheritance hierarchies in two ways. Firstly, by being able to constrain the inheritance hierarchy; secondly, by being able to extend a class in a way that is specific for that class. Nested mixins are a direct consequence of having mixins as attributes. The scope rules for nested mixins are discussed, and shown to preserve the encapsulation of objects.*

1. Introduction

The need to control and to make abstraction of the construction of inheritance hierarchies has been expressed by several authors. On the one hand it seems an obvious extension of the "incremental changes" philosophy of the object-oriented paradigm to be able to incrementally change entire inheritance hierarchies. On the other hand there is a need to control the complexity arising from the use of multiple inheritance [Hendler86] [Hamer92].

A notable example of the first is that given by Lieberman in [Cook87]. The question is how an entire hierarchy of black and white graphical objects can be incrementally changed so that the initially monochrome graphical objects can be turned into coloured objects. In present day systems, one either has to destructively change the root class of this hierarchy by adding a colour attribute, or one has to manually extend each class in the hierarchy with a colour subclass.

The second need stems from the observation 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 different classes [Hamer92]. For example, one would like to put a mutual exclusion constraint on the triangle and rectangle classes, registering the fact that a graphical object can not be both a triangle and a rectangle, and, as such, a class cannot (multiply) inherit from both the triangle and rectangle class.

This article describes a tentative answer to both needs. An extension of mixin-based inheritance [Bracha&Cook90] is proposed. In this extension mixins[1] can be applied dynamically and hence combined flexibly by making use of the available control structures. In addition, the applicability of a mixin can be restricted to a limited, but extensible, set of classes.

The following text presents research results of the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Service - Science Policy Office. The scientific responsability is assumed by its authors.

* The first author is supported by the above research program.

1 To avoid confusion it's important to note here that there is a difference in emphasis in mixins as used in e.g. CLOS and mixins as used in mixin-based inheritance. In the former a mixin is "mixed in" by means of multiple inheritance and linearization, whereas in the latter a mixin is explicitly applied to a class. We will come back to this in the next section.

Mixins provide the extra abstraction that is needed to be able to construct an entire inheritance hierarchy in a building block fashion. The statement that mixins provide exactly the right building blocks for constructing inheritance hierarchies is supported by the results obtained in [Bracha&Cook90], where it is shown that mixin-based inheritance subsumes the different inheritance mechanisms provided by Smalltalk, Beta and CLOS. In contrast with [Bracha&Cook90] the emphasis of our work is on the dynamic applicability of mixins.

Mixins, as in CLOS for example, add to the solution of the first problem, while adding to the problems of the second. Mixins allow unanticipated combinations of behaviour to be made [Stein,Lieberman,Ungar89]. But, when uncontrolled, one faces an explosion of possible combinations of mixins [Lang&Pearlmutter86]. A mechanism to control this combinatorial explosion is needed.

Essential to our approach is that each class knows how to extend itself. A class is asked to extend itself by sending it a message. Each class responds to a limited set of "mixin messages". "Mixin methods" correspond to mixin messages. A mixin method implements an extension for a certain class. Of course mixin methods can be inherited and "late binding" also applies to mixin methods. So, extension of a class is obtained in a very object-oriented way.

A related issue is the dichotomy between encapsulated and non-encapsulated inheritance [Snyder86]. We will show how the nesting of mixin-methods gives rise to a kind of nested scoping that addresses this issue. Typical for this kind of scoping is that visibility of identifiers is *not* directed towards the visibility of variables that are shared by different objects, but rather towards the visibility of inherited variables in the class hierarchy. This difference can best be illustrated by the observation that in an object-oriented programming language it is possible to have two sorts of "omnipresent" variables: 1) a Smalltalk-like global variable, 2) a variable declared in the root class of the class hierarchy. The former is a variable shared by all objects in the system, the latter amounts to an instance variable that is present in each object in the system.

The first part of the paper introduces all concepts in a language independent way. In the second part these concepts are illustrated by means of examples written in Agora. Agora is an object-oriented language under development. One of its hallmarks is the consistent use of mixins for inheritance. Although typing is considered important in Agora, the examples are presented in an untyped version of Agora.

The structure of the paper is as follows. In section 2, we introduce mixin-based inheritance, and the necessary terminology for the rest of the paper. In section 3 we describe mixin-attributes and address encapsulated/non-encapsulated inheritance in this context. Section 4 shows how mixins are introduced in Agora. Section 5 provides a thorough discussion on the scope rules of nested mixins. Section 6 describes how classes are seen as attributes, and the difference between class nesting and mixin nesting. In section 7, the full power of mixin-methods and dynamic application of mixins is explored. The status of mixin-methods is discussed, and a preliminary evaluation is given in section 8. Section 9 discusses related work. Section 10 concludes.

2. Mixin-based Inheritance

In a simplified form[2], inheritance can be modeled as a hierarchical incremental modification mechanism [Wegner&Zdonik88]. A parent P (the superclass) is transformed with a modifier M to form a result R = P + M (the subclass); the result R can be used as a parent for further incremental modification.

Typically the parent, result and modifier are collections of named attributes. From the viewpoint of the result R, the attributes defined in the parent P are referred to as the "inherited" attributes. Attributes defined in the modifier M are referred to as the "proper" attributes of R. Furthermore, in object-oriented languages where encapsulation is promoted, two sorts of attributes exist: private, or encapsulated attributes, and public attributes. Public attributes can be accessed freely; access to the encapsulated attributes is restricted to (the body of) the definition of the public

2 One important aspect we do not address here is the deferred binding of self-reference in inheritance.

and encapsulated attributes. On top of this, in class-based languages we can distinguish class-attributes from instance-attributes[3].

An attribute, either public or private, is selected by name. The result of this can be, amongst others, a computed value, or side effect (e.g. in the case of a method), or simply a stored value (e.g. in the case of an instance variable), or a combination of these, depending on the type of attribute. Each type of attribute can have its own attribute selection rule.

The result R is truly an extension of the parent P (to contrast with e.g. aggregation). Access to the inherited attributes in R is exactly the same as access to the proper attributes of R, though the proper attributes of R take precedence over the inherited attributes in case of name conflicts.

The above incremental modification model of inheritance is a simplification. In most common object-oriented languages, modifiers themselves, also have access to the attributes of the parent being modified. For example, a subclass can invoke operations defined in the superclass (hereafter called parent operations). To model this, a modifier M is parameterized by a parent P that can be referred to in the definitions of the attributes of M. The actual parent is supplied to a modifier when a modifier is composed with a parent. Composing a parent P and a modifier M now takes the form $P \Delta M = P + M(P)$, where the modifier M is no longer a simple set of attributes, but is now a function from a parent to a set of attributes.

The above model is the essence of the model of inheritance in [Bracha&Cook90] where it is used as a basis for the introduction of mixin-based inheritance. It is shown that mixin-based inheritance is a general inheritance model subsuming the inheritance mechanisms provided in Smalltalk, Beta and CLOS.

Whereas in classical single or multiple inheritance the modifier M has no existence on its own (generally it is more or less part of the result R); the essence of mixin-based inheritance is that the modifier M is 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 practice a mixin does not have its base class as an 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 via a pseudo-variable. 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”.

In order to build a new class, first, a mixin is defined; then, this mixin is applied to a base class. One and the same mixin can be used to specialize different (unrelated) base classes. A typical example is that of a colour mixin which adds a colour attribute and the associated accessor methods, and can be applied to classes as different as vehicles and polygons. A typical example involving the invocation of parent operations is that of a “bounds” mixin that constrains the movements of points. The move method attribute defined in the bounds mixin checks for the boundary conditions, and relies on the base class to actually move the point. The actual base class can be a class that implements points by means of cartesian coordinates, or one that implements points by means of polar coordinates.

```
class-based  inheritance
class R
 inherits P
 extended with NamedAttribute1 ... NamedAttributen
endclass

mixin-based  inheritance
M is mixin
    defining NamedAttribute1 ... NamedAttributen
endmixin
class R1 inherits P1 extended with M endclass
```

3 In the paper we only consider mixins as class attributes.

```
class R2 inherits P2 extended with M endclass
```

If it were not for this usage of parent operations the application of a mixin to a base class could easily be mimicked by multiply inheriting from both the base class and the mixin. The parameterization of a mixin with a base class is what makes mixin-based inheritance fundamentally different from multiple inheritance, with the exception of for example CLOS where the inheritance graph is linearized.

A mixin in CLOS is a class that has no fixed superclass and as such can be applied to ("mixed in") different superclasses. In CLOS terminology, this means that a mixin class can do a Call-Next-Method, even though it has no apparent superclass. Mixin-classes in CLOS depend directly on multiple inheritance, and more specifically linearization. Contrary to this, in our work, a mixin is not a class (a mixin can not be instantiated for example), and multiple inheritance is a consequence of, rather than the supporting mechanism for, the use of mixins. In contrast with CLOS, in which mixins are nothing but a special use of multiple inheritance, mixins are promoted as the sole abstraction mechanism for building the inheritance hierarchy.

Mixin-based inheritance gives rise to explicitly linearized inheritance. The order in which mixins are applied is significant for the external visibility of public attribute names. Attributes in the mixin override base class attributes with the same name. In absence of any name conflict resolution mechanism, attribute name look up is determined by application order.

Given a suitable name conflict resolution mechanism mixin-based inheritance can be used to mimic (most forms of) multiple inheritance. With multiple inheritance one and the same class can be reused different times as a parent class in different combinations with other parent classes; with mixin-based inheritance one and the same mixin can be reused in different combinations with other mixins (or base classes). The ability to form "chains" of mixins is appropriate in this case. An evaluation of how mixin-based inheritance addresses multiple inheritance problems is beyond the scope of this paper. The reader is referred to [Bracha92] and [Bracha&Cook90].

Apart from this explicit linearization, duplication of sets of attributes of shared parent classes (mostly used for duplication of instance variables) 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. The inability of, for instance, graph-oriented multiple inheritance to control the duplication of shared parent classes has been shown to lead to encapsulation problems [Snyder86].

3. Mixins as Attributes

Applying the orthogonality principle to the facts that we have mixins and that a class consists of a collection of named attributes, one must address the question of how a mixin can be seen as a named attribute of a class. The adopted solution is that a class lists as mixin attributes all mixins that are applicable to it. The mixins that are listed as attributes in a certain class can only be used to create subclasses of that class and its future subclasses. Furthermore, a class 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 class has the effect of extending that class with the attributes defined in the selected mixin. So, rather than having an explicit operation to apply an arbitrary mixin to an arbitrary class, a class is asked to extend itself.

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 class). A mixin can be made applicable to more or less classes according to its position in the inheritance tree. The higher a mixin is defined the more class 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 class.

```
inheritance of a mixin-attribute
--- Root class attributes ---
 ColourMixin is mixin
   defining colour
 endmixin

 CarMixin is mixin
   defining enginetype
 endmixin

 Car is class obtained by CarMixin extension of Root
 --- class Car inherits ColourMixin defined in the Root class
 ColouredCar is class obtained by ColourMixin extension of Car
```

Note that classes can be attributes too. The meaning is analogous to having a plain object as attribute; there need not be a special relation between a class that is an attribute of a containing class and its containing class. Typically, much more meaning is attributed to nesting of classes [Madsen87]. This will be considered later.

3.1 Applicability of Mixins

What defines applicability of a mixin to a class ? There is no decisive answer to this question. The possible answers accord to the possible varieties of incremental modification mechanisms (e.g. behavioural compatible, signature compatible, name compatible modification, and modification with cancellation) used for inheritance [Wegner&Zdonik88]. If nothing but behaviour compatible modifications are allowed, then only the mixins that define a behaviour compatible modification of a class are applicable to that class.

To put it another way, restricting the applicability of mixins puts a constraint on the possible inheritance hierarchies that can be constructed. The desirability of constraining multiple inheritance hierarchies has already been noted [Hendler86] [Hamer92]. One such constraint is a mutual exclusion constraint on subclasses. The following example is taken from [Hamer92].

Consider a Person class with a Female and a Male subclass. A mutual exclusion constraint on the Female and the Male subclasses expresses the fact that it should not be possible to multiple inherit from Female and Male at the same time. In terms of mixin-based inheritance, we have a Person class, with two mixin-attributes: Female-Mixin, and Male-Mixin. Once the Female mixin is applied to the person class, the Male mixin should not be applicable to the resulting class, and vice versa. This mutual exclusion constraint is realized simply by canceling the Male-Mixin in the Female-Mixin, and by canceling the Female-Mixin in the Male-Mixin.

```
mutual exclusion constraint on classes
--- MarriedPerson class attributes ---
 Female-Mixin is mixin
    defining husband
    canceling Male-Mixin
 endmixin

 Male-Mixin is mixin
    defining wife
    canceling Female-Mixin
 endmixin
```

This solution relies on the ability to cancel inherited attributes. 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 the fact that two mixins are mutually exclusive. Classifiers [Hamer92] play this role for class-based (non mixin-based) languages. A similar mechanism is imaginable for mixins. The reason why the example is given without resorting to such a declarative construction is to show that mixins provide a good basis — and a better basis than classes — to express this sort of constraints on the inheritance hierarchy.

3.2 Mixins and Encapsulated Inheritance

In most object-oriented languages a subclass can access its superclass in two ways. Firstly, by direct access to the private attributes of the superclass (direct access to the implementation details). Secondly, by access to the public attributes of the superclass (parent operations). A mixin is applicable to a class if this class provides the necessary private and public attributes for the implementation of the mixin. This puts an extra restriction on the applicability of a mixin.

The tradeoff between direct access to the implementation details of a superclass and using parent operations is discussed in [Snyder87]. If a mixin depends directly on implementation details of the class it is applied to, then modifications to the implementation of the base class can have consequences for the mixin's implementation. A mixin that uses parent operations only is likely to be applicable to a broader set of classes (it is more abstract). Mixins that make use of the implementation details of a superclass are said to inherit from their superclass in a non-encapsulated way; mixins that make use of parent operations only are said to inherit from their superclass in an encapsulated way

One solution to this problem is to have all superclass references made through parent operations. This implies that for each class, two kinds of interfaces must be provided: a public interface destined for classes (= instantiating clients) that use instances of that class and, a so called private interface for future subclasses (= inheriting clients).

The solution we adopt is to differentiate between mixins that don't and mixins that do rely on implementation details of the base class they are applied to, recognizing the fact that in some cases direct access to a base class's implementation details is needed. To put it differently: a mixin is applicable to a class if this class provides the necessary private attributes for the implementation of the mixin, but not all mixins that are applicable to a class need access to the private attributes of that class (for example the above colour mixin). Essentially, mixins are differentiated by how much of the implementation details of the base class are visible to them. As we will show the solution relies heavily on the ability to inherit mixins. The degree to which a mixin has access to the implementation details of a base class is solely based on whether this mixin is either a proper or rather an inherited attribute of this base class.

Consider a class C that was constructed by application of a mixin MC to a given base class. There are two sorts of mixins that can be used to create subclasses of C: mixins that are proper attributes of C (defined in the mixin MC) and inherited mixins. A mixin that is a proper attribute of the class C, has, by definition, access to the proper private attributes of that class C, and to the same private attributes that the mixin MC has access to. An inherited mixin has no access to the proper private attributes of the class it is applied to. Note that this leads naturally to, and is consistent with, nested mixins. For a mixin to be a proper attribute of the class C, it must be defined in (and consequently nested in) the mixin MC. According to lexical scope rules, it then has access to the names of the attributes defined in the mixin MC.

```
nested mixin-attributes
--- BaseClass attributes ---
 MC is mixin
   defining
    properToC      --- e.g. an instance variable
    PMC is mixin
       defining
           --- properToC is visible here
    endmixin --- PMC ---
 endmixin  --- MC ---

 NotPMC is mixin
   defining
    --- properToC is NOT visible here
 endmixin --- NotPMC ---

C is class obtained by MC extension of BaseClass
PC is class obtained by PMC extension of C
NotPC is class obtained by NotPMC extension of C

--- both PC and NotPC are subclasses of C; only PC has access to C proper attributes ---
```

So, the amount of detail in which a subclass depends on the implementation aspects of its superclass is determined by the relative nesting of the mixins used to create the sub- and superclass. Not only are a mixin's proper instance variables visible for the method declarations in that mixin, but also those of the surrounding mixins. A mixin can be made more or less abstract according to its position in the inheritance tree.

Complete abstraction in mixins can be obtained by not nesting them in other mixins (i.e. defining all mixins on the root class), resulting in a totally encapsulated form of inheritance, as is proposed in [Snyder87]. If abstraction is not required, exposure of inherited private attributes in mixins can be obtained by making the nesting and inheritance hierarchy the same, i.e. by nesting the mixin provided to create the subclass in the mixin provided to create the superclass. This corresponds to Smalltalk-like inheritance.

Of course, combinations between full and no nesting at all are possible. The higher in the hierarchy a mixin is defined, the more objects that can be extended with this mixin, the more abstract the mixin has to be.

A brief comparison between nested mixins and encapsulated inheritance where only parent operations are used to access a superclass, is in place here. In the latter it is the superclass that takes the initiative to determine how much of the implementation will be exposed to inheritors by differentiating between private (visible to inheritors) from public (visible to all) attributes. No distinction is made between inheritors that do make use of the exposed implementation and inheritors that don't. With nested mixins there *is* a distinction between subclasses that do and subclasses that don't rely on a superclass's implementation details. Exposure of implementation details to an inheritor is on initiative of both the ancestor and the inheritor !

4. Mixin-based Inheritance in Agora

4.1 Some Agora Syntax

Agora syntax resembles Smalltalk syntax in its *message-expressions*. We have unary, operator and keyword-messages. Message-expressions can be imperative (statements) or functional (expressions). For clarity, keywords and operators are printed in italics.

`aString` *`size`*	unary message
`aString1` *`+`* `aString2`	operator message
`aString` *`at`*`:index` *`put`*`:aChar`	keyword message

A second category of message-expressions is the category of *reify messages*[4]. Reify messages have the same syntax as message expressions; in the examples they are differentiated from message expressions by having bold-styled keywords/operator. Reify expressions collect all "special" language constructs in one uniform syntax (comparable to lisp special forms). They correspond to syntactical constructs such as assignment statements, variable declarations and many other constructs used in a more conventional programming language. Reify expressions help in keeping Agora syntax as small as possible. Special attention must be paid to the precedence rules. Reify expressions have, as a group, lower precedence than regular message expressions. In each category unary messages have highest precedence, keyword messages have lowest precedence.

`a` **`<-`** `3`	assignment reifier
`c` **`define`**	variable declaration reifier
`c` **`define:`** `3`	same, but with initial value

Message-expressions can be grouped to form blocks.

---

4 In a reflective variant of Agora it is possible to add reifier methods, hence the name. Reifier methods are executed 'at the level of the interpreter' in which all interpreter parameters (context and such) are 'reified'.

```
[c1 define: Complex new ;
 c2 define: Complex new ;
 c1 real:3 imag:4 ;
 c2 <- c1]
```

4.2 Mixins & Methods

The following is an example mixin method. This method adds a `colour` attribute and its access methods to the object it is sent to. In all the examples that follow, mixin definitions standing free in the text (top-level mixins), are presumed to be defined on the root class called `Object`. So, in the example below, the root class `Object` is extended with colour attributes by invoking its *`addColour`* mixin (sending the message *`addColour`* to it). The resulting `ColourObject` class is a subclass of class `Object`.

```
addColour Mixin:
            [ colour define ;
              colour:newColour Method:[colour <- newColour] ;
              colour Method: colour
            ] ;
ColourObject define: Object addColour
```

`Object` is extended with an instance variable "`colour`" and two methods: an imperative method *`colour:`* and a functional method *`colour`*. The body of a method can be either a block or, as can be the case for functional methods, a single expression. To the left of the **Method:** reifier keyword is the pattern to invoke the method; it has the form of an ordinary message expression, except that it has no receiver and the arguments to the keywords are replaced by the names of the formal arguments.

5. Introducing Block Structure in Object-Oriented Languages

Most object-oriented languages define the scope of identifiers more or less ad hoc. In those languages (including Smalltalk), scope rules do not emerge from nesting. Rather, a different look up strategy is defined for each kind of "variable". Smalltalk, for example, offers a blend of variables (class variables, class instance variables, global variables, pool variables, instance variables, arguments, local variables, block arguments) each with its own visibility rule.

While designing Agora, we were aiming to unify all these variants of scoping and to define a simple, uniform strategy to describe the scope of an "identifier". Agora is a block structured language. Blocks and nested structures have come into disfavour in object-oriented languages (with the notable exceptions of Simula and its descendant BETA). Block structures provide locality. The lack of locality in e.g. Smalltalk, where all classes reside in one flat name space, has its drawbacks to structure large programs. Block structures are a natural way to hierarchically structure name spaces. Accordingly scope rules can be imposed. In Agora the visibility of an identifier is solely based on the relative nesting of the block in which this identifier is declared. Nesting results from the declaration of mixins within mixins, methods within mixins, and usage of blocks within methods (for e.g. control structures). Hence, no a priori distinction is made between local variables and e.g. instance variables, nor is there any special provision to declare a global or class variable.

Introducing block structure in an object-oriented system is a very delicate operation [Buhr&Zarnke88]. This is because the "natural" form of scoping that emerges from the nesting of blocks -- identifiers declared in some context are visible in blocks declared in the same context -- can seriously interfere with the notion of encapsulation.

Scope rules can be seen as a mechanism to structure name spaces, whereby a name space is defined as a collection of identifiers with the same scope. One must take care, however, since in an object-oriented language in which objects are considered to be encapsulated, this encapsulation implies that each object has a separate name space;

similarly strictly encapsulated inheritance implies that each sub-object[5] within an object has a separate name space. As is shown earlier, the scope rules for nested mixins structure the name space within a single object. The intention is to regulate the sharing of name spaces of sub-objects. While this breaks the encapsulation of sub-objects, objects are still considered as totally encapsulated, i.e. access to the encapsulated part of an object is reserved to the implementation of the public part of that object, but one sub-object can access the encapsulated part of another sub-object within the same object (mediated by the aforementioned rules).

In the following section we focus on name space sharing for sub-objects. Sometimes there is a need to share name spaces between objects, rather than sub-objects. The above mentioned class variables and global variables, as found in Smalltalk, are examples of such name spaces shared by a number of (or all) objects. In the same way that the scope rules for nested mixins regulate the sharing of name spaces of sub-objects, it is obvious that another set of scope rules can regulate the creation of shared name spaces for objects. This is normally what is accomplished with nested classes in other work, and will be discussed in the section on class nesting versus mixin nesting.

5.1 An Example of Mixin Nesting in Agora

As said before, a mixin is either nested in another mixin, or not nested at all, to control the amount of detail to which a subclass depends on the implementation of a superclass. This is illustrated in the two following examples.

The general idea in the first example is to have turtles which are, in our case, a sort of point that can be moved in a "turtle-like" way (no drawing is involved at the moment). The essence is that a turtle user does not manipulate the location and heading of the turtle directly but uses the `home`/`turn`/`forward` protocol.

```
--- root-class (Object) attributes ---
MakeTurtle Mixin:
  [ location define: Point rho:0 theta:0*pi ;
    heading define: 0*pi ;
    position Method: location ;
    home Method:
           [ location <- Point rho:0 theta:0*pi; heading <- 0*pi ];
    turn:turn Method: [heading <- heading + turn] ;
    forward:distance Method:
           [ location <- location +
                             (Point rho:distance theta:heading) ] ;

    MakeBounded Mixin:
     [ bound define: Circle m:location r:infinite ;
       home Method:
            [ bound <- Circle m:location r:infinite ; super home ] ;
       newBound:maxRho Method:
            [ bound <- Circle m:location r:maxRho ] ;
       forward:distance Method:
            [ newLocation define ;
              newLocation <- location + (Point rho:distance
                                           theta:heading) ;
              (newLocation - (bound center)) rho > bound r
                  ifTrue:
                    [super forward:
                          (((LineSeg p1:location p2:newLocation)
                                 intersect:bound)  - location)rho ]
                  ifFalse: [ super forward:distance ] ] ] ] ;
```

5 Each object is composed out of sub-objects according to the inheritance hierarchy.

```
Turtle define: Object MakeTurtle ;
BoundedTurtle define: Turtle MakeBounded ;
aBoundedTurtle define: BoundedTurtle new ;
aTurtle define: Turtle new ;
aBoundedTurtle forward:1 ;
aTurtle forward:3
```

Once the turtle is defined, the next step is to create a subclass that puts boundaries on the movements of the turtle. In the example turtles are restricted to move within the bounds of a circle. For this purpose the `forward` method is overridden in the subclass that implements this boundary checking. This overridden `forward` method uses direct access to the turtle instance variables `location` and `heading` in its implementation.

For the construction of the classes `Turtle` and `BoundedTurtle`, two mixins, `MakeTurtle` and `MakeBounded` respectively, are defined. To make sure that the class `BoundedTurtle` inherits from class `Turtle` in a non-encapsulated way, the `MakeBounded` mixin is nested in the `MakeTurtle` mixin. Notice that, since the `MakeBounded` mixin is defined only for `Turtle`, it can only be used to extend the `Turtle` class and its subclasses. Not only is it impossible to extend the root class `Object` with the `MakeBounded` mixin since it is not defined for the root class but also since `Object` does not define the `location`/`heading` instance variables that are required by the `MakeBounded` mixin.

Each instance of Turtle and each instance of `BoundedTurtle` has its own set of `location`/`heading` instance variables. Furthermore, if in the `MakeBounded` mixin an instance variable were to be declared with a name that collides with a name in the `MakeTurtle` mixin (e.g. an instance variable with the name "`heading`"), then each `BoundedTurtle` would have two instance variables with this name. One instance variable would only be visible from within methods defined in the `MakeTurtle` mixin, the other instance variable would only be visible from within methods defined in the `MakeBounded` mixin. There is a "hole in the scope" of the instance variable defined in the `MakeTurtle` mixin. So, there is no merging going on for instance variables with equal names, neither is it an error to have an instance variable with the same name in a subclass (as is the case in Smalltalk). Notice that identifier lookup is a static operation: The instance variable that is referred to in an expression can be deduced from looking at the nested structure of the program. No dynamic lookup strategies are applied. Similar observations can be made for non-nested mixins. Encapsulating the names of instance variables in this way is an important aid in enhancing the potential for mixin composition. This is all the more important if mixins are used to create/emulate multiple inheritance hierarchies.

Thus, if a mixin is nested in another mixin, classes created by the innermost mixin are always (not necessarily direct) subclasses of classes defined by the outermost mixin. However, the reverse statement is not always true. Nesting is not a requirement for subclasses.

```
--- root-class (Object) attributes ---
MakeDrawingTurtle Mixin:
  [ penDown define: true ;
    togglePen Method: [penDown <- penDown not] ;
    forward:distance Method:
           [ newPosition define ;
             oldPosition define: self position ;
             super forward:distance ;
             newPosition <- self position ;
             penDown ifTrue:
                  [… draw line from old position to new position …]
           ] ;
      MakeDashed Mixin:
        [ dashSize define: 1 ;
          setDashSize:newSize Method: [dashSize <- newSize] ;
          forward:distance Method:
             [penDown
                ifTrue:
                   [ 1 to: (distance div: dashSize)
                         do:    [ super forward: dashSize ;
                                  self togglePen ] ;
                     super forward: (distance mod: dashSize) ;
```

```
                        penDown <- true
                    ]
                ifFalse: [ super forward:distance ]
            ]
        ]
    ] ;

DrawingTurtle define: Turtle MakeDrawingTurtle ;
DashedDrawingTurtle define: DrawingTurtle MakeDashed ;
```

The goal in the above example is to extend the `Turtle` class so that it draws, or does not draw (depending on the status of the pen), on the screen where the turtle is heading. The drawing capabilities can be added fairly independently of the implementation of the turtle. Once again the `forward` method is overridden. But all that is needed in the implementation of the overridden `forward` method is the old `forward` method and a method that returns the current location of the turtle. Notice that, even though the location of the turtle must now be made public (to read), the `heading` instance variable is still encapsulated. The `MakeDrawingTurtle` mixin that implements this extension does not have to be nested in the `Turtle` mixin, resulting in a `MakeDrawingTurtle` mixin that can be applied to other sorts of turtle classes that respect the `forward`/`position` protocol.

Earlier on we said that the `MakeBounded` mixin could only be applied to the `Turtle` class and its subclasses. `DrawingTurtle` is such a subclass. We now have two ways to create bounded drawing turtles. On the one hand, by applying the `MakeDrawingTurtle` to a `BoundedTurtle` (`DrawingBoundedTurtle` **`define:`** `BoundedTurtle` *`MakeDrawingTurtle`*), on the other hand, by applying the `MakeBounded` mixin to a `DrawingTurtle` (`BoundedDrawingTurtle` **`define:`**`DrawingTurtle` *`MakeBounded`*). In this example both results are the same; the `forward` method in the `MakeDrawingTurtle` mixin is such that it only draws a line up to the position where the turtle has moved, even if it moved a shorter distance than was intended.

It is important to note that the order of mixin application has no effect on the exposure of implementation details of the applied mixins to each other. This is important since this greatly enhances the reusability of mixins. The order in which the mixins *`MakeDrawingTurtle`* and *`MakeBounded`* are applied has no effect on the respective exposure of implementation details of the turtle base class to the inheriting clients `DrawingBoundedTurtle` or `BoundedDrawingTurtle`. The `makeBounded` and the `makeDrawingTurtle` cannot access each other's encapsulated part (independently of which mixin is applied first), and in both cases only the `MakeBounded` mixin has access to the `turtle` class's implementation details. It is coincidental in the example that we can choose in which order the mixins *`MakeDrawingTurtle`* and *`MakeBounded`* are applied, and that both results exhibit the same *behaviour*. In cases where this choice is not available, any dependence of the exposure of implementation details on the order of mixin application would seriously restrict the reuse of mixins.

6. Classes as Attributes, Class Nesting Versus Mixin Nesting

Even though the emphasis, up until now, was put on mixins as attributes, a class can also be considered as an attribute of an object. The main difference with having mixins as attributes (and classes as attributes in non mixin-based inheritance) is that this does not introduce a new set of scope rules. Since the definition of a class is nothing but the application of a mixin to a base class, having classes as attributes does not imply nested classes, as illustrated below.

```
makeA Mixin:
  [ B define: Object makeB ; -- local class
    ...
  ] ;
makeB Mixin:
  [...] ;

A define: Object makeA
```

In this example, a class `A` is defined, that keeps a reference to a local class `B`. The description of `B` however is not nested in `A`, since `B` is created with a mixin defined on the root class `Object`. Normal scope rules apply to the identifier B.

In most object-oriented languages, classes reside in a name space shared by all objects. Indeed, in most cases a class should be visible for all objects. Due to the lack of global variables in Agora this can only be realized by inheriting attributes that refer to classes, rather than by having a global name space for classes. Classes that should be globally visible must be defined as attributes of the root class. Since all classes are derived from the root class, and the mixins to derive these classes are de facto nested in this root class, each class inherits and has direct access to these attributes. Of course this results in each object having all globally visible classes as instance variables. In a practical implementation this need not be a problem.

For non mixin-based inheritance the ability to have classes as attributes normally implies class nesting. The major difference between nesting of classes and nesting of mixins lies in their respective relation to encapsulation. As was amply discussed, nested mixins respect the encapsulation of objects.

When nesting classes, the nesting is used to create shared name spaces for objects. Although nested classes can be very useful (as is shown in both [Madsen87] and [Buhr&Zarnke88]), this mechanism can be used by a programmer to break the encapsulation of objects. The next example is an example of class nesting. Both b1 and b2 can access the same variable "i" (instance variable of class A), i.e. they share the same variable "i" in their name space. Modification of this variable in, let's say b1, has an effect on the variable seen by a and b2. Instance variables (i.e. the variable "i") of an instance of class A (i.e. a) can directly be accessed by instances of class B (i.e. b1,b2) , even if there is no relation (sub or super) between these two classes.

```
class A extends SuperOfA
 i : Integer ;
 class B extends SuperOfB
  -- i is visible here !
 end B ;
end A ;
a : A ;
b1 : a.B ;
b2 : a.B
```

The two different forms of scoping (nested mixins and nested classes) do not mix very well. This is apparent when one uses class nesting and a non encapsulated form of inheritance at the same time (as is the case in BETA). Then, identifier look up is ambiguous, because in every class, two different contexts can be consulted: the surrounding block context or the context of the superclass. In the above class nesting example this ambiguity would be apparent if an instance variable with the name “i” were defined in the superclass of B. This problem is resolved by giving priority to one of both name spaces in case of a name conflict, e.g. by first looking in the superclass chain and then in the surrounding scope (here again the superclass chain must be searched and so on …).

7. Mixin Methods

The fact that mixin application is realized by mere message passing, and that mixins can be applied dynamically has clear advantages. In this section we will give a simple example of dynamic mixin application, an example of late binding of mixins, and the role of the self pseudo-variable in mixin-methods.

Mixins can be combined to form chains of mixins that can be applied as a whole. Chains of mixins are useful to abstract over the construction of complex class hierarchies. A simple example is given making use of the Turtle classes shown earlier on. The idea is to construct different sorts of dashed drawing turtles without having to explicitly create a simple drawing variant, and a dashed drawing variant for each sort of turtle. This is, of course, the simplest example of how dynamic mixin application is used to abstract over the construction of an inheritance hierarchy.

```
MakeDashedDrawing Method: self MakeDrawingTurtle MakeDashed ;

DashedDrawingTurtle define: Turtle MakeDashedDrawing ;
DashedDrawingBoundedTurtle define: BoundedTurtle MakeDashedDrawing
```

In the example a chain of mixins is constructed as a method that successively applies two mixins. A declarative operator (as in [Bracha&Cook90]) to construct chains of mixins (or even entire hierarchies) could prove useful.

To illustrate the use of late binding of mixin attributes, consider a program in which two freely interchangeable implementations of point objects exist; one implementation based on polar coordinates and one based on cartesian coordinates. In some part of the program, points must be *locally* (for this part of the program only) restricted to bounded points, i.e. points that can not move outside given bounds. To do this, every point must have a mixin attribute to add methods and instance variables that implement this restriction. Each of the point implementations can have its own version of this mixin in order to take advantage of the particular point representation. For example, the mixin defined on polar coordinate represented points, can store its bounding points in polar coordinates in order to avoid excessive representation transformations. An anonymous point class (one of which we don't know whether it is a polar or a cartesian point; typically a parameter of a generic class) can now be asked to extend itself to a bounded point by selecting the bounds mixin by name. The appropriate version will be taken.

```
MakeCartesianPoint Mixin:
  [ x define: 0 ; y define: 0 ;
    move:aPoint Method: … ;

    MakeBounded Mixin:
            [ bound define: CartesianBasedBounds new;
              move:aPoint Method: …
            ]
  ] ;



MakePolarPoint Mixin:
  [ rho define: 0 ; theta define: 0*pi ;
    move:aPoint Method: … ;

    MakeBounded Mixin:
            [ bound define: PolarBasedBounds new;
              move:aPoint Method: …
            ]
  ] ;

--- suppose Point is bound to either a Polar or Cartesian Point
BoundedPoint define: Point makeBounded
```

The `self` pseudo-variable plays an important role in mixin methods. The receiver of a message that caused the execution of a mixin is the class that is being extended. Motivated by convenience, but somewhat different from the meaning of the self pseudovariable in a method and from what could be expected, the self pseudo variable used in the execution of a mixin-method contains a reference to the new class that is being defined (rather than to the old base class that is being extended). The contents of this variable can be stored for later use by instances of this class. Instances of a class often create other instances of the same class (e.g. recursive data structures). In a class-based language this is simply done by referring to the class's name. Since a mixin can be applied to a set of different base classes, there is no single name it can refer to.

Consider the definition of the cartesian and polar point classes. Suppose a method to add points was not included in the definition of these classes, and you do not want to destructively change these classes to add this method. What normally should have been defined in some abstract superclass of the two different point classes, you now want to define as an afterthought. The obvious thing to do is to make a mixin that adds the addition method so that it can be applied to both classes. The idea is to let the result of the addition method be a point of the same kind as the receiver point. This can be done by making use of the self pseudo variable in the mixin method that adds this extra behaviour. The contents of the self pseudo variable is used as initial value for the instance variable `AddablePoint` so that every instance of class `AddablePolarPoint` / `AddableCartesianPoint` has an instance variable referring to the class itself.

```
MakePointAddable Mixin:
  [ AddablePoint define: self ;
    sum:argument Method: AddablePoint x:(…) y:(…)
  ] ;

AddableCartesianPoint define: CartesianPoint MakePointAddable ;
AddablePolarPoint define: PolarPoint MakePointAddable
```

Note that the meaning of the self pseudo variable in a mixin-method is different from (and also has a slightly different purpose than) a construct such as "self class" in Smalltalk. In a situation where the sum message is sent to a subclass `X` of e.g. `AddablePolarPoint`, with the above construct the sum method still yields instances of class `AddablePolarPoint`. In a construct where a Smalltalk-like "self class" would be used this would yield instances of the subclass `X`.

8. Status, Evaluation, and Future Work

An implementation of Agora exists, built on top of Smalltalk. Agora is an experimental language, hence the implemented language differs on some points from what is described in the paper. However, mixin-methods are implemented according to the description given in the paper.

One important aspect omitted in this paper is the reflective architecture of Agora. This is an important issue since some of the features that one expects to find built into the programming language, we intend to introduce by making use of reflective facilities (in the style of [Kiczales,des Rivières&Bobrow91], [Jagannathan&Agha92]). This includes, but is not limited to, features such as resolution of name collisions, abstract methods, declarative combination and mutual exclusion of mixins. Experiments in this direction are under way.

Experience with the use of mixin-methods in Agora is somewhat limited to small scale experiments. The performance, both in time and memory, of the Smalltalk implementation is such that no large experiments have been attempted. Still some observations can be made.

Although the scope rules seem unfamiliar, newcomers (in our limited experience, of course) to Agora quickly take up the scope rules. The choice of the relative nesting of mixins is seen as an incentive to think about how abstract a mixin must be made.

Another observation is that due to the use of mixin-methods, different functionalities that are normally found in one class tend to be split up into different mixins. One could say that a class tends to be split up into different "views" or "perspectives". Also, there is a trade-off between overriding of mixin-methods and overriding of "ordinary" methods, i.e. a generic print method can be defined by overriding this method in each possible class, or by having a print mixin that is overridden in each class. The latter then corresponds to a printing view on each class. It has the advantage that otherwise unrelated classes can be extended with the same printing "view". The net effect is that programs are written by first defining all kinds of mixins, and then combining these mixins to classes as needed. Larger experiments are needed to further determine how mixin-methods will be used in practice.

It should be stressed that we think the importance of mixin-methods is in their ability to express otherwise tangled inheritance hierarchies in a more intelligible way. Central to this is the notion of applicability of mixins. In its current form, however, the applicability of a mixin can only be based on the absence or presence of implementation details in possible base classes. Further work to see how other sorts of constraints on the applicability of mixins can be (declaratively) expressed, is needed.

9. Related Work

Our work is an extension of mixin-based inheritance as was introduced in [Bracha&Cook90]. To their work we add dynamic application of mixins, mixins as attributes and the resulting scope rules for nested mixins. The extra polymorphism gained by viewing mixins as attributes seems to us an important enhancement to mixin-based inheritance. In contrast with [Bracha&Cook90] the mixin-methods used in Agora remain untyped at the moment.

The relation to nested classes [Buhr&Zarnke88][Madsen87] has been discussed above. The correspondence between so called virtual superclasses [Madsen&Møller-Pedersen89] in BETA, and mixins has already been noted [Bracha92]. The same remarks as in the previous paragraph apply to the relation between mixin-methods and virtual superclasses.

Agora was primarily designed as a prototype-based language. Mixin-based inheritance can also be applied to prototype-based programming languages. New prototypes are 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. Here as well, extensions can be considered as separate abstractions. The terminology mixins and mixin-application from the class-based case can be retained.

Once again, we must consider the fact that a mixin can be seen as an attribute of an object (or an instance attribute of a class). In this case too, an object (the prototype) is extended by selecting one of its mixin-attributes. So, in contrast with other prototype-based languages (e.g. Self [Ungar&Smith87]) an object plays a more "active" role in the extension process. This is especially important if one considers the possibility to override inherited mixin-attributes. The full extent of combining mixins and prototypes is outside the scope of this paper.

Having mixins as instance attributes is very similar to "enhancements" described in [Hendler86]. We agree that being able to associate functionality with instances rather than classes has several advantages. The advantages of dynamic classification have also been discussed in the classifier approach of [Hamer92]. Both approaches lack the equivalent of late binding of mixin attributes. Although our approach lacks the equivalent of having classifiers as first-class values (which would amount to first class mixin "patterns").

10. Conclusion

Mixin-methods are proposed as a uniform framework to control and make abstraction of the way inheritance hierarchies are constructed. Central to this are the notions of applicability of mixins and dynamic application of mixins. Due to the treatment of mixins as attributes, mixins can be inherited and overridden. This introduces an extra level of abstraction in the way classes are extended that is not available (to the authors' best knowledge) in present day object-oriented languages.

The scope rules for nested mixins were discussed. On the one hand having mixins as attributes naturally leads to nested mixins, on the other hand an important factor in the applicability of a mixin to a base class is the availability of the necessary implementation details in this base class. The scope rules for nested mixins integrate both concerns. They are such that an Agora programmer has a fine-grained control over the amount of implementation details of the base class a mixin has access to.

While similar issues have been discussed elsewhere, no work has been found that discusses all of the above issues in one single framework.

11. Acknowledgments

The paper benefited from discussions with Franz Hauck and Michel Tilman. The authors thank Karel Driesen and Serge Demeyer for giving useful comments on an earlier version of the text, and Geert Bollen for proofreading. We also thank the anonymous referees for their helpful comments.

12. References

[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.

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

[Buhr&Zarnke88] P.A. Buhr, C.R. Zarnke. Nesting in an Object-Oriented Language is NOT for the Birds. In Proc. of ECOOP‘88 European Conference on Object-Oriented Programming, pp.128-143, Springer-Verlag 1988.

[Cook87] S. Cook. Panel Varieties of inheritance. In OOPSLA’87 Addendum to the proceedings, pp.35-40, ACM Press 1987.

[Hamer92] J. Hamer. Un-Mixing Inheritance with Classifiers. In Proc. 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 Proc. of Object-Oriented Programming Workshop 86, Sigplan Notices Vol 21 (10), pp.98-106, October 1986.

[Jagannathan&Agha92] S. Jagannathan, G. Agha. A Reflective Model of Inheritance. In Proc, of ECOOP'92 European Conference on Object-Oriented Programming, pp.350-371, Springer-Verlag 1992.

[Kiczales,des Rivières&Bobrow91] G. Kiczales, J. des Rivières and D.G. Bobrow. The Art of the Meta-Object Protocol. MIT Press, 1991.

[Lang&Pearlmutter86] K. J. Lang and B. A. Pearlmutter. Oaklisp: an Object-Oriented Scheme with First Class Types. In Proc. of ACM Conf. on Object-Oriented Programming, Languages, and Systems, pp.30-37, ACM Press 1986.

[Madsen87] O. L. Madsen. Block Structure and Object-Oriented Languages. Research Directions in Object-Oriented Programming B. Shriver and P. Wegner (eds), pp 113-128, MIT Press 1987.

[Madsen&Møller-Pedersen89] O. L. Madsen B. and B. Møller-Pedersen. Virtual Classes, A powerful mechanism in object-oriented programming. In Proc. of ACM Conf. on Object-Oriented Programming, Languages, and Systems, pp.397-406, ACM Press 1989.

[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.

[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.

[Ungar&Smith87] D. Ungar & R. B. Smith. Self: The Power of Simplicity. In Proc. of ACM Conf. on Object-Oriented Programming, Languages, and Systems, pp 227-242, 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 Proc. of ECOOP‘88 European Conference on Object-Oriented Programming, pp.55-77, Springer-Verlag 1988.