Agora: Message Passing as a Foundation for Exploring OO Language Concepts

Wim Codenie, Koen De Hondt, Theo D'Hondt, Patrick Steyaert

Programming Technology Lab (PROG/WE)
Brussels Free University (VUB)
Pleinlaan 2
B-1050 Brussels, Belgium
agora@prog.vub.ac.be

abstract

Agora is a framework for exploring object-oriented languages. In this paper we will discuss a particular prototype-based instance of Agora that features a general mixin-based approach to (multiple) inheritance. One of the major innovations of Agora is that many important features of object-orientation including inheritance, slot access, reification, cloning and inline objects are introduced by means of message passing rather than by ordinary programming structures.
The syntax and semantics of Agora is explained and illustrated with examples. Furthermore, an FTP-site is given where a Smalltalk implementation of Agora can be found.

1 Introduction

Agora is solidly rooted in the object-oriented paradigm. Agora is a *prototype*-based language [Ungar&Smith87] featuring a generalised *mixin* [Bracha&Cook90][Steyaert&al.93] approach to inheritance. The extension of prototypical objects through the application of mixins is embedded in the lexical scoping of identifiers in Agora [Buhr&Zarnke88, Madsen89]. Consistent *reification* [Smith84] is the approach used for capturing features such as name binding, deferred evaluation, self reference, etc.

Any of these features, taken by themselves, do not constitute innovations. Agora, however, is innovative because these features are bound together in one consistent language framework with message passing as its most important driving force. Whereas in most programming languages, e.g. inheritance, name binding mechanisms, cloning of objects, variable syntax, inline objects, etc. are expressed in structures that differ fundamentally from ordinary programming structures, Agora requires but one programming paradigm (message passing) for all the components of the system.

Mixins are specified as methods and mixins are applied in the same way that ordinary messages are sent. Reification is equally structured as message passing: Reifiers are nothing but methods defined within the bodies of abstract grammar prototypes. Access to instance variables and methods is unified by the introduction of receiverless patterns that perfectly fit in the message passing paradigm. Cloning methods are suited to combine the creation and initialisation of new objects. Agora's single slot nested objects (SSNO's) are in many ways superior to 'conventional' Smalltalk blocks.

In this paper we present the syntax of Agora and describe a kernel of reifiers which turns it into a usable programming language. The implementation of Agora is not just an implementation of the reifiers that will be described in this paper, but rather the realisation of a more general framework of which this particular variant is only a concretisation. Agora, as presented here, is not an aim in itself, but rather a means to express our ideas about message passing. It is possible – and not at all difficult – to build flavours of Agora where the emphasis is laid on topics different from those discussed here (e.g. reflection, concurrency, class-based systems,...). However, this is beyond the scope of this paper. For more information we refer the reader to [Steyaert94].

Section 2 explains the syntax of Agora, while sections 3,4,5,6,7 describe the kernel. We assume that the reader has a basic knowledge of object-oriented programming. In particular we assume that concepts like instance variables, methods, inheritance, late binding, etc. are familiar. An elaborate example is given in section 8. The last section briefly discusses the implementation and gives an FTP-site, where an implementation of Agora can be found.

2 Agora Syntax

Agora syntax resembles Smalltalk syntax in its *message expressions*. The different kinds of message expressions are: Unary, operator and keyword messages. Message expressions can be imperative (statements) or functional (expressions).

```
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 *receiverless messages*. Receiverless messages have the same syntax as the pattern part of message expressions. Their principal usage is to invoke messages on an implicit receiver, for example to invoke local methods; they will also be used as part of other syntactical structures where message patterns need to be manipulated (i.e. method declarations).

```
size                                  Receiverless unary message (identifier).
+ aString                             Receiverless operator message.
at:index put:aChar                    Receiverless keyword message.
```

A third category of message expressions is the category of *reify messages*. Reify (receiverless) messages have the same syntax as (receiverless) message expressions except for their bold-style keywords/operators. Reify expressions (i.e. reify messages, receiverless reify messages, and reify aggregates as can be found in the next paragraph) collect all "special" language constructs in one uniform syntax (comparable to Lisp special forms). They correspond to syntactical constructs such as variable declarations, pseudo variables, control structures and many other constructs used in a more conventional programming language. Reify expressions help in keeping the Agora syntax as concise as possible.

```
self                                  Receiverless unary reifier.
a <> 3                                Operator reifier.
a variable: 3                         Keyword reifier.
```

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. Parentheses are used to break the default precedence rules.

```
dict at:dict size put:3 + 10 sqrt           = dict at:(dict size) put:(3 + (10 sqrt))
at:index mixin method:self basicAt: index   = ((at:index) mixin) method:(self basicAt: index)
```

Message expressions can be grouped to form blocks. Blocks are an example of the third kind of reify expressions, i.e. reify expression where the delimiters are the variable part of the syntax (it is not necessary to have bold styled delimiters since delimiters are not used for any other purpose). Although other expression aggregates are imaginable, in this text only blocks will be considered. A special kind of block is the empty block denoted by ' [ ] '.

```
[  c1 variable:Complex clone ;
   c2 variable:Complex clone ;
   c1 real:3 imag:4 ;
   c2 <- c1 ]
```

The concrete grammar of Agora in BNF form can be found in appendix B.

Agora's syntax consists of two layers. The above given syntax only specifies the generic or variable layer. Reifiers form the variable part of Agora's syntax. Much of the design of Agora is found in the exact list of reifiers that can be used by the programmer. In the next sections we will discuss the set of reifiers currently supported by our implementation.

3 Slots

Agora objects consist of an *encapsulated* (or *local*) and a *public* part. The encapsulated part of an object contains all the attributes that are private to an object (= attributes that can only be accessed from within the object), while the public part defines the object's interface with the external world. In section 8 we will illustrate how the encapsulated and public part of an object are related to each other. For now, it suffices to know that they exist.

Both the encapsulated and public part of an object contain *slots* (or *attributes*). Basically we can distinguish between two kinds of slots: *Variable* and *method* slots. Variable slots are used to store values and are declared with a variant of the **variable** reifier. By default, variables are declared in the encapsulated part of the object. The **variable** reifier can only be used in combination with receiverless unary patterns (i.e. identifiers). Its three variants are listed below.

`x` **`variable`**	Variable declaration in the encapsulated part.
`x` **`variable:`** `3`	Same, but with initial value.
`d` **`<>`** `Dictionary`	Same, but with clone of initial value.

A method slot, on the other hand, contains an expression (not necessarily a block) that must be evaluated in the context of the object on which the method slot is defined. The **method**: reifier is used to declare methods. By default, the methods are declared in the public part of the object. This reifier can only be used in combination with receiverless patterns. If a keyword or operator pattern is used, the arguments must be unary patterns, representing the names of the formal arguments of the method. These names can then be used in the body of the method. The following is the declaration of a method with pattern 'increment:' in the public part of an object. The body of the method increments the value of a not further specified local variable.

```
increment:amount method:counter <- counter + amount
```

It is also possible to define variables and methods in respectively the public and encapsulated part of an object by inserting the appropriate slot specification reifier (**public** or **local**) in the declaration of the slot.

`x` **`public variable`**	Declaration of a variable accessible through the public part.
`m:n` **`local method:self`** `n`	Declaration of a method accessible through the encapsulated part.
`y` **`public local <>`** `p`	Declaration of a variable accessible through the public and encapsulated part.
`n` **`local public method:`**`[]`	Declaration of a method accessible through the public and encapsulated part.

If both the **public** and **local** reifier are used, a slot is declared that can be accessed in a local as well as in a public way. It is important to note that the values of these entries are shared. For example, an assignment to the public variable 'y' in the above example has an effect on the local variable 'y' and vice versa.

Slots must be accessed and modified through message passing. In case of an access to a local slot, however, the receiver must be left implicit. So, for example, a local variable 'x' is accessed via the receiverless unary pattern 'x' and its value is set to e.g. 3 with the receiverless keyword message 'x: 3'. An equivalent assignment can be used too:

`x`	Variable retrieval.
`x:3`	Variable assignment.
`x` **`<-`** `3`	Assignment reifier, equivalent to the above.

Access to a local method follows the same syntax. Therefore it is impossible to know whether one is accessing a variable or a method by only examining the call site of a message expression. We have deliberately chosen for this resemblance because it enhances the uniformity of the accessor protocol of objects. For example, it is very easy to replace a variable by a method with the same behaviour and vice versa.

`x`	Receiverless unary message activation.
`x:3 y:4`	Receiverless keyword message activation.
`+ 10`	Receiverless operator message activation.

Note that since receiverless messages are used to access local slots, and there is no receiver specified, receiverless messages, in contrast to other messages, obey the lexical scoping when being looked up (more on this in section 8). Also keep in mind that it is *not* possible to assign values to arguments of methods or mixins.

Access to a public slot of an object is realised by sending a message with the object as a receiver. Again, the same syntax is used for variables and methods.

`x m:10`	Send the message 'm:' with argument 10 to the object obtained by evaluation of 'x'. 'm' can be either a public method or a public instance variable.

Thus, when a variable 'x' is declared (either local or public), two entries are added. The first entry 'x' is a functional slot that can be used to retrieve the value of variable 'x'. The other slot 'x:' has an imperative behaviour: It stores the argument in the variable. There are two slot specification reifiers (**functional** and **imperative**) that are used to make the distinction between functional and imperative behaviour.

```
x variable                          Define a local variable 'x' that can be accessed in a functional and imperative way.
x functional imperative variable    Same as above.
x functional public variable:10     Define a public slot that can only be read (similar to a constant).
x imperative variable:10            Define a slot that can only be assigned.
```

The reifiers **functional** and **imperative** can also be used in combination with the method declaration reifier to restrict the application possibilities of methods. If omitted, the declared method can be used in a functional as well as in an imperative way.

```
at:index functional method:[...]                          Declare a method that can only be invoked functionally.
at:index put:value imperative method:[...]                Declare a method that can only be invoked imperatively.
ifTrue:trueObject ifFalse:falseObject imperative functional method:[...]   Same as default.
```

If the **functional** or **imperative** slot specification reifier is used with the **local** and **public** reifier, the functional or imperative behaviour is associated with *both* the local and public entry. Thus it is not possible for the moment to declare a variable that can only be read through the public interface and assigned through the local interface. This feature, which would be very useful, does not pose any technical problems in our implementation. The reason why we have not introduced it up to now is that it is not easy to find an elegant reifier notation for it.

If a method is used in a functional context, its body must of course return a value. This can be accomplished by using a functional expression as body. A block is made functional by using the **return** reifier. This reifier identifies the value that has to be returned when the evaluation has reached the end of the block. It is important to note that this reifier does not alter the control flow within the block (as is the case for the return statement in Smalltalk). The following is an example of a functional block.

```
[   a variable ;
    a:2 + 3;
    a return ]
```

4 Control Structures

Due to Agora's nature control structures can be introduced in two ways. The first is as user defined control structures based on a notion comparable to first class block contexts (e.g. Smalltalk) or closures (e.g. Scheme). A second way is by the definition of reifiers that implement a fixed set of control structures. The following list shows all the control structures that are implemented as reifiers.

```
x = y    ifTrue:[a: 2; b: 3]   ifFalse:x n:10.
x = y    ifFalse:[a: 2; b: 3]  ifTrue:x n:10.
x = y    ifTrue:a d: 10.
x = y    ifFalse:a d: 10.
a < 10   whileTrue:a:a + 1
false    whileFalse:a:a + 1
counter  for:1 to:10 do:counter print.
         Evaluate the expression behind 'do:' in the current receiver with counter bound to 1 .. 10.
```

A operator reifier **#** is provided to create *single slot nested objects* (SSNO's). The public part of such an object contains one single method slot. The local part of the object is the encapsulated part of the object in which the SSNO is nested. The receiver of the **#** reifier is the single pattern to which the resulting SSNO will respond. The argument is an expression that defines the body of the single method. As is usual for methods, the body is not evaluated until the pattern used on the left-hand side of the **#** reifier is sent to the resulting single method-slot object. The evaluation takes place in the encapsulated part of the object in which the single slot object is defined. Note that once a SSNO is created, it can be manipulated like any other object. Therefore, SSNO's are very useful for implementing control structures (cfr. Smalltalk block contexts). Unary patterns are used for SSNO's that have no arguments; keyword and operator patterns are necessary to define arguments. Consider the following example. The method of the SSNO in the variable 'block' is not evaluated until the messages 'block value:10' and 'block value:20' are evaluated. Note that it is not possible to use slot specification reifiers (e.g. functional, imperative, ...) in combination with the pattern of the **#** reifier.

```
[   y variable:1;
    block variable;
    block:(value:x # (x + y) print);
    block value:10;             ->      11.
    block value:20 ]            ->      21.
```

Agora has a **self** pseudo variable reifier and a reifier to invoke parent operations.

self	self pseudo variable reifier.
(at:3 put:5) **super**	super message invocation reifier; invokes the parent's ' at:put:' method.

The **self** pseudo variable reifier, when evaluated, returns the current receiver. When a method is locally invoked by a receiverless pattern, **self** remains the same object. This behaviour is desirable because the actual receiver of the method remains unchanged. When, on the other hand, a method is invoked in a public way, the **self** pseudo variable is bound to the receiver. The **super** reifier delegates its message argument (which must be receiverless) to the parent object. The **super** reifier has a somewhat unusual form. Rather than having a super pseudo variable, invocation of parent operations looks more like a control structure. The difference between a super pseudo variable and a parent invocation control structure is subtle, however.

Note also the difference between receiverless messages and messages to the self pseudo variable. Both are directed to different parts of the receiver. The former is used to invoke local operations the latter to invoke public operations. So '**self** x' and 'x' generally have a different result.

5 Mixin methods

An Agora program is organised as a hierarchy of nested blocks with one single root block. The execution of the program starts with the evaluation of this root block in a predefined object. As will become clear later, every object created by the program will be an extension of this predefined root object.

The following is an example of an Agora program with two simple mixin methods: 'colourMixin' and 'carMixin'. A mixin is a method declared with the slot specification reifier **mixin**. The body of a mixin method must always be a block. The mixin 'colourMixin' adds a public 'colour' attribute to the object it is sent to. In all the examples that follow, mixin definitions declared in the top-level block are defined on the root object. So, in the example below, the root object is extended with a colour attribute by invoking its 'colourMixin' mixin (sending the message 'colourMixin' to it). The resulting 'colouredObject' object is an inheritor of the root object. As can be seen from the example it is possible to access the colour attribute of a coloured object (because the attribute is declared public). In an analogous way, it is also possible to send the mixin 'carMixin' to the root object. We can even go a step further and define coloured cars by first sending the 'car' mixin followed by the 'colour' mixin. The resulting object ('colouredCar') now has colour and car attributes.

```
[  colourMixin mixin method:[colour public variable ] ;
   carMixin mixin method:
     [ brand public variable ;
       owner public variable ] ;
   colouredObject variable:self colourMixin ;
   colouredObject colour:"blue" ;
   colouredObject colour print ;
   car variable:self carMixin ;
   car owner:"John" ;
   car brand:"Trabant" ;
   colouredCar variable:self carMixin colourMixin ;
   colouredCar colour:"red" ;
   colouredCar owner:"John" ;
   colouredCar brand:"Ferrari" ]
```

The above example is a good illustration of how (multiple) inheritance hierarchies can be constructed with mixin methods. Each mixin has its own functionality (i.e. colouring) and mixins can be freely combined to form more complex objects. A more thorough discussion of mixin methods in Agora can be found in [Steyaert&al.93]. The implications of the nesting of mixins are briefly discussed in section 8.

In Agora, mixins and methods are very similar. Methods are to be considered as executing in a temporary, local extension of the receiver object. An explicit notion of closures, or method activations can be avoided due to the object-based nature of Agora. The difference between methods and mixins is that the one extends the receiver object only temporarily and the other extends the receiver object permanently. Due to this similarity, arguments and local variables of methods can be defined and accessed in a totally similar way to instance variables. This opens the question of method declarations local to method declarations. Agora's design restricts all declarations within method bodies, to variable declarations !

The fact that mixin application is realised by mere message passing, and that mixins can be applied dynamically has clear advantages. 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 object hierarchies. A simple example is the definition of a method to create coloured cars.

```
colouredCar method: self colourMixin carMixin;
...
ColouredCar variable: self colouredCar ;
```

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 co-ordinates and one based on Cartesian co-ordinates. 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 co-ordinate represented points, can store its bounding points in polar co-ordinates in order to avoid excessive representation transformations. An anonymous point object (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.

What differentiates this variant of Agora from most object-oriented programming languages is the highly expressive combination of nested mixin methods and object-based programming (apart from the reflective architecture, which will be discussed later on).

6 Object Creation

Agora objects are created by taking copies (clones) of existing objects. In its most elementary form this takes the form of a clone reifier. A more elaborate cloning method will be discussed later. Whereas in class-based languages one talks about classes and instances, in object-based languages one speaks about *prototypes* and *clones*. By convention the names of objects that are consistently used as prototypes start with an uppercase letter.

`p <- P clone`	Assign a copy of 'P' to 'p'.
`p <> P`	Declare a new variable 'p' with initial value a copy of 'P'.

Another way to create new objects is by application of mixins to objects. Since mixins can be applied to objects (rather than classes), different independent extensions of one and the same object can be made.

```
[  person mixin method:[ name public variable ] ;
   sportsMan mixin method:[ number public variable ] ;
   student mixin method:[ number public variable ] ;
   Person functional variable:self person ;
   john <> Person ;
   johnAsSportsMan variable:john sportsMan ;
   johnAsStudent variable:john student ;
   john name:"john" ;
   johnAsStudent number:100 ;
   johnAsSportsMan number:200 ;
   johnAsStudent name print ;          ->      'john'
   johnAsSportsMan name print ;        ->      'john'
   johnAsStudent number print ;        ->      100
   johnAsSportsMan number print ]      ->      200
```

In this example two extensions of a same, shared parent ('john') are made. These extensions implement different views on the same object: A view on 'john' as a student and another view on 'john' as a sportsman. This example encodes, in Agora, the 'multiple viewpoint' example from [Carré&Geib90].

Yet another possibility to create objects is the use of cloning slots. If a variable slot 'a' is declared with the **cloning** slot specification reifier, a functional accessor 'a:' is added besides the functional slot 'a' described earlier. When this message is sent in a functional context, the object is copied and the assignment is executed in this new object. Then, the newly created object is returned. If the cloning reifier is used to declare a method, a functional method is declared so that the body of the method is always executed in a copy of the receiver. The result of the cloning method is the newly copied object. The **cloning** reifier cannot be used in combination with the **imperative** slot specification reifier. Consider as an example the following program for point manipulations:

```
[  point mixin method:
     [ x public local cloning variable ;
       y public local cloning variable ;
       x:xx y:yy method:(x:xx) y:yy ] ;
   Point functional variable:self point ;
   p variable: Point x:10 y:20 ;
   q variable: p x:100 ]
```

Cloning slots are perfectly suited to combine the creation and initialisation of objects. In the above example the value of 'q' is a point derived from 'p', except that its x co-ordinate is set to 100. In a Smalltalk solution to this problem the initialisation message (usually a private method) on the instance level can always be sent to the object to modify the co-ordinates. With cloning slots this is impossible: Every attempt to modify one of the co-ordinates results in the creation of a new point.

7 Lazy evaluation

Sometimes, when declaring variables, it is necessary to delay the evaluation of the right-hand side of the declaration. This is especially true when recursive data structures are defined. In those cases, the slot specification reifier **lazy** can be used. In the following example a local variable 'a' is defined. If it were not declared lazy this program would result in an error because the mixin 'm' does not exist when the declaration is evaluated. Declaring it lazy postpones evaluation until the value of 'a' is needed (in this example in the last statement). The **lazy** slot specification reifier can not be used for method declarations.

```
[  a lazy functional public local variable:self m ;
   m mixin method:[ x public variable ] ;
   a x:10 ]
```

8 An Example of Mixin Nesting in Agora

Whether a mixin is either nested in another mixin, or not nested at all depends on the amount of detail on which an inheritor is dependent for the implementation of its heir. We have already discussed an example of non nested mixins.

The general idea is to have turtles which are, in our case, a sort of point that can be moved in a “turtle-like” way. 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.

Two mixins are defined in the top-level block. First we have a mixin that describes the behaviour of points. A point stores its co-ordinates using polar notation but accessors for Cartesian co-ordinates are also provided. The 'turtle' mixin is the interesting one. It defines, and implements the behaviour of turtles. The mixin can be used to create such turtles (e.g. 'Turtle').

Once the turtle ('Turtle') is defined, the next step is to create inheritors that allow turtles to be drawn on some drawing device. In the example we have omitted the definition for the drawing device. For this purpose the 'forward:' method is overridden in the inheritor that implements the drawing. This overridden 'forward:' method uses direct access to the turtle instance variable 'position' in its implementation.

For the construction of drawable turtles a third mixin 'drawingTurtle' is defined. To make sure that drawing turtles inherit from turtles in a non-encapsulated way, the 'drawingTurtle' mixin is nested in the 'turtle' mixin. Notice that, since the 'drawingTurtle' mixin is defined only for 'Turtle', it can only be used to extend the 'Turtle' prototype and its inheritors. Not only is it impossible to extend the root object with the 'drawingTurtle' mixin, since it is not defined for the root object, but also since the root object does not define the 'position' instance variable that is required by the 'drawingTurtle' mixin.

Each clone of 'Turtle' and each clone of 'DrawingTurtle' has its own set of 'position/heading' instance variables. Furthermore, if in the 'drawingTurtle' mixin a local slot were to be declared with a name that collides with a name in the 'turtle' mixin (e.g. a variable with the name 'position'), then each drawing turtle would have two local slots with this name. One slot would only be visible from within methods defined in the 'turtle' mixin, the other slot would only be visible from within methods defined in the 'drawingTurtle' mixin. There is a “hole in the scope” of the slot defined in the 'drawingTurtle' mixin. So, there is no merging for slots with equal names, neither is it an error to have a slot with the same name in an inheritor (as is the case in Smalltalk). Notice that look up in the local part is a static operation: The receiverless patterns that are referred to in an expression can be deduced from looking at the nested structure of the program. No dynamic look up 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 when mixins are used to create/emulate multiple inheritance hierarchies. Thus, when a mixin is nested in another mixin, objects created by the innermost mixin are always (not necessarily direct) inheritors of objects defined by the outermost mixin. However, the reverse statement is not always true. Nesting is not a requirement for inheriting as can be seen in the coloured car example from section 5.

```
[  pi functional variable:3.14159 ;

   point mixin method:
     [ rho functional public local cloning variable:0 ;
       tetha functional public local cloning variable:0 ;
       rho:r tetha:t  method:(rho:r) tetha:t ;
       x  local public method:(rho * (tetha cos)) truncated ;
       y  local public method:(rho * (tetha sin)) truncated ;
       x:xx y:yy method:
         [   t variable ;
             xx = 0 ifTrue:t:(pi / 2) ifFalse:t:(yy / xx) arcTan ;
             (rho:((xx*xx) + (yy*yy)) sqrt) tetha:t return] ;
       + p functional method:Point x:(x + p x) y:(y + p y) ;
       print method:[ rho print ;tetha print ] ] ;

   turtle mixin method:
     [ position variable: Point x:200 y:100 ;
       heading variable: 0;
       turn:t method: heading:(heading + (t*(pi / 180))) ;
       forward:d method:[position: (position + (Point rho:d tetha:heading))] ;
       drawingTurtle:device mixin method:[
         forward:d method:
           [   old variable: position ;
               (forward:d) super  ;
               device from:old to:position ]]] ;

 Point functional variable:self point ;
 Turtle functional variable:self turtle ;
 DrawingTurtle functional variable:Turtle drawingTurtle:drawingDevice ;
 t  <> DrawingTurtle ;
 i from: 1 to: 12 do:
   [ t forward:30 ;t turn:30 ] ]
```

Figure 1(a) gives the representation of the internal object structure of the 'DrawingTurtle' prototype. The public part is shown on the right side, the local part is on the left.

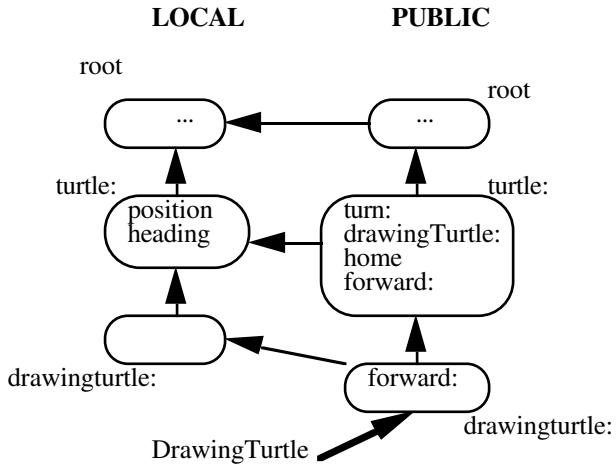

Figure 1(a)

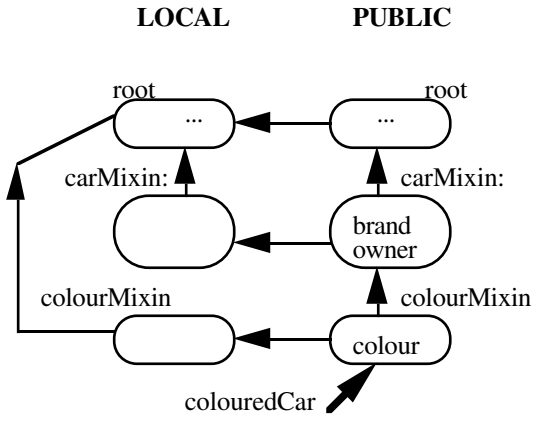

Figure 1(b)

Every mixin application that is used to create the object introduces a new segment in the both the local and public part of the object. Public segments are linked to each other according to the chain of mixins that is applied to obtain the object. Local segments are linked according to the nesting of the mixins. If the body of a mixin is nested in another mixin, the segment of the innermost mixin points to the segment of the more global mixin. In this example the segments are similarly linked in the public and local part. This is because the 'drawingTurtle' mixin is nested in the 'turtle' mixin. No linking appears between local segments that correspond to not nested mixins. This is illustrated in figure 1(b) where the object representation of the 'colouredCar' object from section 5 is given. Finally, each public segment is linked to its corresponding local segment. During method look up, when a method is found in a public segment, its body is evaluated in the corresponding local segment.

9 The Agora implementation

An experimental implementation of Agora has been made in Smalltalk, more precisely in VisualWorks[1]. It must be considered as an open design of which this variant is only an instance [Steyaert94]. The implementation has a reflective architecture. It is possible to add new reifiers to the system from within an Agora program. This is realised by the introduction of a **reifier**: reifier that is similar to the **method**: reifier. New object structures can also be defined and installed in the interpreter. Discussion of these features is beyond the scope of this paper. For more information we refer the reader to [Steyaert94].

You can download the Agora implementation (and related papers) from the following FTP-site: `progftp.vub.ac.be`. The directory to search for is 'agora'. Agora comes with a complete installation manual and many examples (including the examples from this paper) that illustrate the power of mixins. If you have any comments and/or suggestions, please feel free to email them to `agora@prog.vub.ac.be`.

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.

[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.

[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.312-321, ACM Press 1990.

[Goldberg&Robson89] A. Goldberg, and D. Robson:**Smalltalk-80, The Language**, Addison-Wesley Publishing Company, Reading Massachusetts, 1989.

[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.

[Smith84] B.C. Smith: **Reflection and Semantics in Lisp**, Conf. Rec. of the 11th ACM Symposium on Principles of Programming Languages, pp. 23-35, January 1984.

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

[Ungar&Smith87] D. Ungar and R. B. Smith: **Self**: **The Power of Simplicity**, In Proceedings of OOPSLA'87 Conf. pp. 227-242, ACM Press,1987.

Appendix A: List of Reifiers

All the reifiers listed below are explained in the paper except for the **comment** reifier. When evaluation reaches the comment reifier the evaluation of the receiver is ignored.

```
"This is a comment" comment
```

Agora Reifiers					
Declaration reifiers:	**variable**	**variable**:	**method**:	**<>**	
Slot specification reifiers:	**local**	**public**	**functional**	**imperative**	
	mixin	**lazy**	**cloning**		
Control structures:	**ifTrue**:	**ifFalse**:	**ifTrue**:**ifFalse**:	**ifFalse**:**ifTrue**:	
	whileTrue:	**whileFalse**:	**from**:**to**:**do**:	**super**	**self**
Miscelanous:	**clone**	**comment**	**return**	**#**	**<-**

[1] Visual works is a trademark of ParcPlace Systems, Inc.

Appendix B: Agora Grammar in BNF form

Terminals are included in quote (“”) symbols. Production rules have the form: … -> …, where the left-hand side is always a non terminal. In the right-hand side of a production vertical bars (|) are used to indicate alternatives, square brackets ([]) to indicate optional parts, and curly brackets ({}) to indicate zero or more repetitions.

Agora Concrete Grammar

```
Expression             -> ReifierMessage | ReifierPattern | Pattern

ReifierPattern         -> ReifierUnaryPattern | ReifierOperatorPattern
                                | ReifierKeywordPattern
Pattern                -> UnaryPattern | OperatorPattern | KeywordPattern
ReifierMessage         -> ReifierOperation { ReifierKeywordPattern }
ReifierKeywordPattern  -> BoldKeyword ReifierOperation
ReifierOperation       -> ReifierUnary [ ReifierOperatorPattern ]
ReifierOperatorPattern -> BoldOperator ReifierOperation
ReifierUnary           -> Message { ReifierUnaryPattern }
ReifierUnaryPattern    -> BoldIdentifier
Message                -> Operation { KeywordPattern }
KeywordPattern         -> Keyword Operation
Operation              -> Unary [ OperatorPattern ]
OperatorPattern        -> Operator Operation
Unary                  -> Factor { UnaryPattern }
UnaryPattern           -> Identifier
Factor                 -> Literal | Aggregate | "(" Expression ")"
Aggregate              -> LeftAggregateSymbol
                          [ Expression { ";" Expression } ]
                          RightAggregateSymbol
Identifier             -> Character { CharacterOrDigit }
BoldIdentifier         -> BoldCharacter { BoldCharacterOrDigit }
Operator               -> OperatorSymbol [ OperatorSymbol ]
BoldOperator           -> BoldOperatorSymbol [ BoldOperatorSymbol ]
Keyword                -> Identifier ":"
BoldKeyword            -> BoldIdentifier ":"
Literal                -> StringLiteral | RealLiteral |
                          IntegerLiteral | CharacterLiteral
StringLiteral          -> ""Character}""
CharacterLiteral       -> "'Character"'
IntegerLiteral         -> PositiveInteger | "-" PositiveInteger
PositiveInteger        -> Digit {Digit}
RealLiteral            -> Real | RealWithExponent
Real                   -> IntegerLiteral. PositiveInteger
RealWithExponent       -> Real "e" IntegerLiteral
Character              -> "a" | "b" | … | "z"
Digit                  -> "0" | "1" | … | "9"
CharacterOrDigit       -> Character | Digit
BoldCharacter          -> "a" | "b" | … | "z"
BoldDigit              -> "0" | "1" | … | "9"
BoldCharacterOrDigit   -> BoldCharacter | BoldDigit
OperatorSymbol         -> ">" | "<" | "|" |"\" | "*" | "+" | …
BoldOperatorSymbol     -> ">" | "<" | "|" |"\" | "*" | "+" | …
LeftAggregateSymbol    -> "[","{"
RightAggregateSymbol   -> "]","}"
```