Chapter 4

Specialising the Framework with Inheritance

4.1 Introduction

In the previous chapter we introduced and discussed a framework for an object-based programming language. In this chapter we will specialise this framework to include inheritance. It will be shown that inheritance can be added without modifying the basic structure of the framework.

First we will discuss the issues involved in designing a language that employs some form of inheritance. Closely related to this is the topic of scoping and visibility rules in object-oriented languages. An overview of the design issues will be given. Based on this a full-fledged object-oriented programming language is presented (Agora). It is shown that although Agora differs from Simple in some fundamental ways, the framework presented in the previous section can be used as a skeleton to implement Agora. This implementation will be used to define a second layer of abstract classes in the framework that handles inheritance.

4.2 Inheritance, Design Issues

In this section we will discuss the design issues that arise when designing an object-oriented programming language that employs some form of inheritance. We use the term inheritance for both object-based inheritance and class-based inheritance. We already saw that both forms are closely related, they are both a form of incremental modification. The problems and design issues that are discussed are independent of whether one considers either class-based or object-based inheritance. To emphasise this, we will use the terms inheritor or heir (for subclass/inheriting object), and ancestor or parent (for superclass/ancestor object). The terms class and object will be used both, even though in most places 'class or object' would be more appropriate. First we will go into some general problems with inheritance, then we will consider the problems that relate strictly to *multiple inheritance*.

4.2.1 Inheritance and Encapsulation Problems

In a similar way that it is important for an object that it can encapsulate attributes, it is equally important that an inheritor can encapsulate the fact that it depends on an ancestor for the implementation of certain attributes. When an inheritor is not free to change its inheritance structure we say that the inheritance is exposed. It is important that inheritance is not exposed to future inheritors or clients that use instances of a class/object. Examples of exposed inheritance will be given in the section on multiple inheritance, but can also be found in typed object-oriented languages where the notions of type and class are identified with each other.

Another interaction between encapsulation and inheritance stems from the fact that in most object-oriented languages an inheritor can access its ancestor in two ways. One, by direct access to the private attributes of the ancestor (direct access to the implementation details). Two, by access to the public attributes of the ancestor (parent operations). The trade-off between direct access to the implementation details of an ancestor and using parent operations is discussed in [Snyder87].

If an inheritor depends directly on implementation details of its ancestor, then modifications to the implementation of the ancestor can have consequences for the inheritor's implementation. An inheritor that uses only parent operations is likely to be less sensitive to ancestor changes (it is more abstract). Inheritors that make use of the implementation details of an ancestor are said to inherit from their ancestor in a non-encapsulated way; inheritors that make use of parent operations only are said to inherit from their ancestor in an encapsulated way. That is to say, whereas strict encapsulation implies that the private attributes of an object are not directly accessible by other objects; similarly *strictly encapsulated inheritance* implies that the private attributes of a subobject[1] within an object are not directly accessible by other subobjects within that object.

One solution to this problem is to have all ancestor references done through parent operations. This implies that for each class/object two kinds of interfaces must be provided: a public interface destined for instantiating clients that use instances of that class/object and, a so called private interface for future inheritors that are called the *inheriting clients* of the class/object.

1 Each object is composed out of subobjects according to the inheritance hierarchy.

4.2.2 The Need for Flexible and Controllable Inheritance

The need to control and abstract over the way inheritance hierarchies are constructed has been expressed in several ways and in several places. 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 triangle and rectangle classes, registering the fact that a graphical object can not be both a triangle and a rectangle. Consequently, a graphical object class can not multiply inherit from both the triangle and rectangle class.

Multiple inheritance is used to combine existing classes in order to construct new classes. *Tangled inheritance hierarchies* occur in multiple inheritance hierarchies where the classes expose a high degree of possible combinations. This is often the case where classes are decomposed in an unusually fine granularity or where classes are decomposed into different views or perspectives. An example of a tangled multiple inheritance hierarchy is given below. The example shows points that can be implemented as either polar or cartesian points. Point movements can be bounded; both the cartesian and the polar point classes have their own way to implement these boundaries.

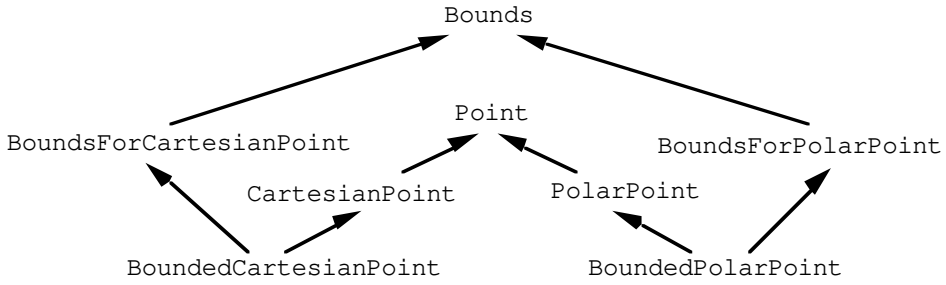

Figure 4.1

Some observations concerning tangled hierarchies can be made on the basis of the above hierarchy. The first observation is that there is a proliferation of classes. In the worst case, where there are two orthogonal sets of classes that can be combined, the possible subclasses are in the cartesian product of these two sets. In the example this is not the case since it does not make sense to combine e.g. BoundsForCartesianPoint with PolarPoint. The two sets of classes in the example are not orthogonal. In most languages with multiple inheritance all combinations of orthogonal sets of classes must be explicitly constructed. Most of the so constructed classes are so called *empty classes*, i.e. they contain no declarations. Empty classes serve only as a basis for instantiation. This phenomenon is referred to as the *proliferating subclass problem*.

Another observation that can be made is the fact that the above hierarchy does not represent all the information that is available about this hierarchy. Essentially this hierarchy only shows what combinations have been made but says nothing about what other possible combinations can still be made, or, equally important, what combinations can not be made. The fact that it is senseless to combine e.g. BoundsForCartesianPoint with PolarPoint is not represented in the hierarchy.

Even more important than the observation that constraints on the possible combinations are not represented, is the observation that the hierarchy does not represent the fact that the classes BoundsForCartesianPoint and BoundsForPolarPoint play a similar role in the respective resulting combination classes BoundedCartesianPoint and BoundedPolarPoint. The correspondence between the two can be summarised in the following table:

	CartesianPoint	PolarPoint
Add Boundaries	BoundsForCartesianPoint	BoundsForPolarPoint
Result	BoundedCartesianPoint	BoundedPolarPoint

Figure 4.2

Multiple inheritance mechanisms are poor at expressing the role an ancestor plays in the inheriting class; let alone to express the fact that two classes, used each as ancestor for a different inheritor, play a similar role (such as in the above example) for their respective inheritors.

4.2.3 Multiple Inheritance

It is important to distinguish single inheritance from multiple inheritance. Single inheritance is characterised by the fact that each inheritor has exactly (or at most) one parent. With multiple inheritance each inheritor can have multiple parents. Proponents of single inheritance say that multiple inheritance is not as yet well-understood and that in most cases single inheritance is satisfactory to express their problems. Proponents of multiple inheritance find that a sufficient number of 'real world' problems can not be expressed with a tree-structured classification mechanism.

The design issues in inheritance mostly concern multiple inheritance, since single inheritance is yet well-understood. In the design of an inheritance mechanism the chief concern is the interaction between inheritance and encapsulation, and also how flexible an inheritance hierarchy can be constructed.

In languages that use multiple inheritance each class/object can have multiple parents. This gives rise to *inheritance graphs* rather than *inheritance trees* (as is the case with single inheritance). This graph is a directed acyclic graph. Usually the graph has one single root (i.e. the root class/object). We will discuss different multiple inheritance mechanisms and the different problems involved. These different mechanisms are distinguished by how they treat the inheritance graph and how name conflicts are resolved. We will briefly overview the two major problems involved.

The first problem one has to face when an inheritor inherits from two or more parents is the problem of *name collisions*. Parents can have attributes with the same names. In [Knudsen88] three different sorts of name collision are identified: intended name collision, casual name collision and illegal name collision. These three sorts of name collision correspond roughly to respectively: 1) a name

collision where the attributes of the names that collide are intrinsically the same (e.g. in the case where the names are inherited, in their turn, of a common superclass) or intrinsically separable (e.g. in the case where the attributes are of a different nature such as a method and an instance variable, or the attributes have a different domain such as two methods that apply to arguments with disjunct types); 2) collision of names that are not related at all, but the names are separated by e.g. a qualifier; 3) collision of names that are not related and the names are not separated.

Conflicts are resolved in different ways (according to the specific language at hand). Either a mechanism is provided to rename the conflicting operation in one of the subclasses, or a mechanism is provided for qualified message passing, or it is an error to inherit two operations with the same name. With qualified message passing a message is qualified with an ancestor's name to direct the method lookup.

The second problem one has to face when dealing directly with the multiple inheritance graph is the *diamond problem* (or *common ancestor duplication problem*) depicted in figure 4.3. An inheritor B multiple inherits from classes/objects S1 and S2, which in their turn inherit from a common ancestor A. Now the question is whether the inheritor B will contain one or two subobjects A, and how the name collisions for the attributes defined on A will be be handled. That is, do we deal with the graph as is, or do we transform the graph into a linear chain, or do we transform the inheritance graph in e.g. a tree where both S1 and S2 have their own 'copy' of the ancestor A ?

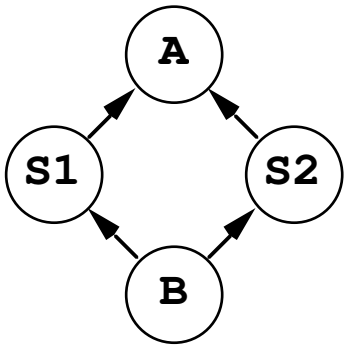

Figure 4.3

We will consider all problems that are related to the diamond problem. We will see how these problems are resolved, or not resolved, in the different multiple inheritance strategies.

Graph Multiple Inheritance

A first approach to multiple inheritance are strategies that deal with the inheritance graph directly without transforming it.

In *graph multiple inheritance* operations are inherited along the inheritance graph unless they are redefined in an inheritor. Conflicts arise when two operations with the same name are inherited along different paths in the graph. Conflicts are resolved either with qualified message passing or with renaming of conflicting attributes.

In general a distinction is made between 1) conflicts arising from two attributes that have the same name and are essentially different and 2) conflicts arising from two attributes that have the same name and are essentially the same (i.e. an attribute that is inherited along two different paths from one and the same non-direct ancestor, such as can be seen in the diamond problem). In general, whereas for the former case special provisions must be made (i.e. it is an error or qualified message passing must be used), the latter case is not really seen as a name conflict. This option is motivated by convenience but is shown to violate the encapsulation of inheritance.

It can be shown easily that not considering these "same attributes" as name conflicts exposes the use of inheritance. Consider the diamond figure again. An attribute 'x' defined on A is, according to the above rule, not conflicting in B. However if S2 is changed so that it is implementing attribute 'x' itself, this operation becomes a conflict in B. On the other hand, if names inherited from a shared parent along different paths are considered as name conflicts then in languages where all classes/objects inherit from a given root class/object (with attributes defined on the root, of course) any instance of multiple inheritance would result in conflicts for the root attributes.

The foremost reason why it is argued that we should not deal directly with the inheritance graph is that it exposes the inheritance structure [Snyder87]. The inheritor B should not be aware of how S1 and S2 are realised and whether they inherit from A or not. Let's take a look at the situation where S1 implements by itself (i.e. by not inheriting from A) all attributes that were previously defined in A and compare it with the situation where S1 inherits from A for the purpose of implementing these attributes. In the former case side-effects on the 'A-attributes' via S1 are not visible to S2, in the latter case they are. Just like in the previous paragraph this means that if one deals with the inheritance graph directly the implementor of S1 has not the freedom to reimplement S1 without inheriting from A or vice versa. The inheritance structure in S1 is exposed to its inheriting clients.

An additional problem related to graph oriented multiple inheritance is that of the *undesired duplicate parent operation invocation*. This will be illustrated with an example (example taken from [Snyder87]). It is the, by now almost classical, example of a point class with its two subclasses 'BoundedPoint' and 'HistoryPoint'. Points can be moved. History points record all point movements. Bounded points can only be moved within certain boundaries.

```
class Point
  instance variables: x y
  methods
    moveX:dx moveY:dy
      x := x + dx
      y := y + dy
    location
      ^(x,y)
endclass

class HistoryPoint
 inherits Point
 extended with
  instance variables: history
  methods
    moveX:dx moveY:dy
      history record: ("moved to: ", self location)
      super moveX:dx moveY:dy
endclass
```

```
class BoundedPoint
 inherits Point
 extended with
  instance variables: bounds
  methods
    moveX:dx moveY:dy
      if (self location + (Point x:dx y:dy)) within: bounds
        then super moveX:dx moveY:dy
endclass
```

In an attempt to make points that are both bounded and keep a history of moves, we define the `BoundedHistoryPoint` class that multiply inherits from `BoundedPoint` and `HistoryPoint` classes. In the definition of the move operation the move operation of both parents must be invoked. Of course this simple solution has the wrong effect that each move message sent to a bounded history point results in two move messages to the point subobject. The `x` and `y` instance variables of the receiving object are incremented twice. The classes `BoundedPoint` and `HistoryPoint` cannot be 'sufficiently' combined with graph multiple inheritance to form bounded history points.

```
class BoundedHistoryPoint
 inherits BoundedPoint HistoryPoint
 extended with
  methods
    moveX:dx moveY:dy
      super BoundedPoint.moveX:dx moveY:dy
      super HistoryPoint.moveX:dx moveY:dy
endclass
```

Linear Multiple Inheritance (implicit and explicit)
Implicit *linear multiple inheritance* strategies first flatten the inheritance graph for each class in a linear chain and then treat the collection of chains as a single inheritance hierarchy. Of course the linearisation strategy must obey some constraints. First of all it must preserve the ordering of classes/objects along each path in the inheritance graph. That is, an inheritor of some class/object may not become an ancestor of that class/object or vice versa during the linearisation process. Other restrictions may apply also. In CLOS, for example, programmers have some degree of control over the linearisation process by the order in which the superclasses of some class are listed; this order is also respected in CLOS's linearisation strategy.

Name conflicts are resolved automatically in the linearisation process. In case of a name conflict one of the conflicting attributes is selected even though there is no single best choice. Consider again the diamond picture from above, and consider a possible linearisation (A—S2—S1—B) of this inheritance graph (figure 4.4). Suppose both S1 and S2 define a conflicting attribute 'x'. In the linearised graph B will inherit the x attribute of S1, and the x attribute of S2 has been 'masked away' by S1, even though in an equally correct linearisation the reverse would be true.

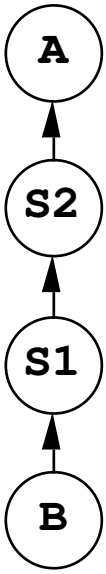

Figure 4.4

Related to this problem is the fact that an inheritor can not reliably communicate with its direct ancestors. Due to linearisation sometimes unrelated classes/objects are inserted between an inheritor and one of its direct ancestors. This is the case for S1 in the linearised inheritance graph above. Even though in the graph representation S1 has A as direct ancestor, in the linearised graph S2 has become the direct ancestor of A (A becomes an indirect ancestor of S1). Especially when name conflicts are involved this can give surprising results.

Another way to look at the above problem is that with linearised inheritance it is not possible to 'combine' two conflicting methods. In case of a conflict there is one single attribute that is visible to the inheritor. The other attributes are masked. To resolve this problem most languages with linearised inheritance provide a declarative mechanism that is called method combination (in CLOS, for example, an entire range of method combination mechanisms are provided [Moon89]). Method combination provides solutions to the above problems for a set of standard situations, still it does not take away the need for a class to be able to set up reliable communication with its direct ancestors.

Apart from the advantage that no name conflicts arise, linearised multiple inheritance performs well with respect to the problem of duplicate parent operations invocation. This is best illustrated by looking at the point, history point and bounded point example from the previous section. Whereas with graph multiple inheritance the classes BoundedPoint and HistoryPoint could not be combined to get the desired effect, this is perfectly possible with linearised inheritance.

In an encoding with linearised multiple inheritance the BoundedHistoryPoint class's move method must only invoke (and in fact *can* only invoke, since no name conflicts can arise) the move method of its ancestor once. This encoding is shown below.

```
class BoundedHistoryPoint
 inherits BoundedPoint HistoryPoint
 extended with
  methods
    moveX:dx moveY:dy
      super moveX:dx moveY:dy
endclass
```

This solution is heavily based on the knowledge that the inheritance graph will be linearised. In a possible linearisation, shown below, the effect will be as such that the move method in BoundedHistoryPoint invokes the one in HistoryPoint which in turn invokes the one in BoundedPoint which in his turn invokes the final move method in the Point ancestor. In fact what we first considered as a problem, i.e. the insertion of unrelated superclasses, now turns out to be part of a solution. We will see below that this sort of techniques can be generalised to what is called *mixin-based inheritance*, and that both the HistoryPoint and the BoundedPoint classes are best encoded as *mixins* (see also [Snyder87]).

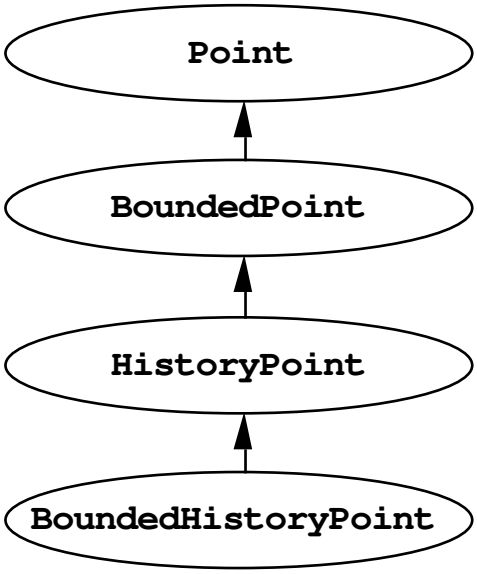

Figure 4.5

Although implicit linearisation solves some of the problems that occur in graph multiple inheritance, these problems are solved in a radical way. No name conflicts can occur due to changes in the inheritance hierarchy of some class (changes that do not alter the semantics of that class), but on the other hand conflicts are resolved even if there is no clear 'best choice'.

An important, and possitive, side effect of the unintended interleaving of an unrelated ancestor class is the concept of *mixin-classes*. Again, let's have a look at the diamond example, and its linearisation (A—S2—S1—B) above. It is possible for S1 to invoke parent operations that are not declared in its direct ancestor A, but due to linearisation are found in the newly assigned ancestor S2 (e.g. consider an operation x that is defined on S2 and not on A that is invoked via a parent operation from S1).

Going one step further, it is trivial to see that it is possible to have classes that have no apparent ancestor but that do invoke parent operations in a meaningful way. This sort of classes have been named mixin-classes since they rely on the linearisation to be 'mixed in' at the appropriate place (i.e. as inheritor from a class that provides the necessary operations) in the linearised inheritance hierarchy. The effect is that it is possible to create mixin-classes that can be

applied (mixed-in) to a set of different superclasses. (in mixin terminology also called base classes) The prototypical example of a mixin-class is the one that adds colour attributes to all sorts of base classes. We will see many more examples in the remainder of this text.

Mixins have been identified as very useful and flexible building blocks to construct inheritance hierarchies. Another approach to multiple inheritance uses mixins as the sole mechanism to create inheritance hierarchies, and is called *mixin-based inheritance* [Bracha&Cook90] [Bracha92] [Hense92] [Steyaert&al.93] [Codenie,Steyaert,Lucas92].

Mixin-based inheritance and its relation to multiple inheritance will be further explored in subsequent sections.

Tree Multiple Inheritance

If it is said that linear multiple inheritance solves name collision problems in a radical way by not having any name collisions at all, then *tree multiple inheritance* solves name collision problems in a radical way by always having name collisions. Tree multiple inheritance is directly motivated by a need to solve problems related with the diamond problem. We saw that graph multiple inheritance exposes the inheritance structure due to two reasons. One, by not duplicating parents that are inherited via different paths in the inheritance graph; two, by not signalling name conflicts when one and the same operation is inherited via different paths in the inheritance graph. With respect to these two problems tree multiple inheritance takes exactly the two opposing design decisions as graph multiple inheritance.

Just as with implicitly linearised multiple inheritance the multiple inheritance graph is transformed, but in a less radical way. Rather than transforming the inheritance graph for some class in a linear chain, the inheritance graph is transformed into a tree where each parent that is inherited via different paths in the graph has been duplicated. For the diamond above this results in the following tree.

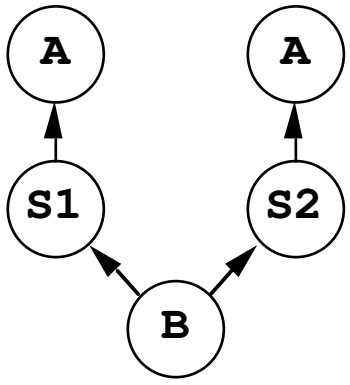

Figure 4.6

After this transformation all name collisions are treated on an equal footing. No name collisions can arise from attributes that are inherited via different paths, because all ancestors in joining paths have been duplicated.

Although tree multiple inheritance solves the inheritance encapsulation problems of both graph and implicitly linearised multiple inheritance it does so at a certain cost. But, let us first look at an example (example due to [Knudsen88]) where tree multiple inheritance works fine.

In the example we want to model a small part of the employees database from some university. There are two sorts of employments, lecturer and administrative staff. Each employee has a seniority. The seniority is used for example to calculate wages. In a very sensible way, the management of the seniority attribute is factored out in some abstract superclass called '`University Employee`'. For the example no other attributes are considered.

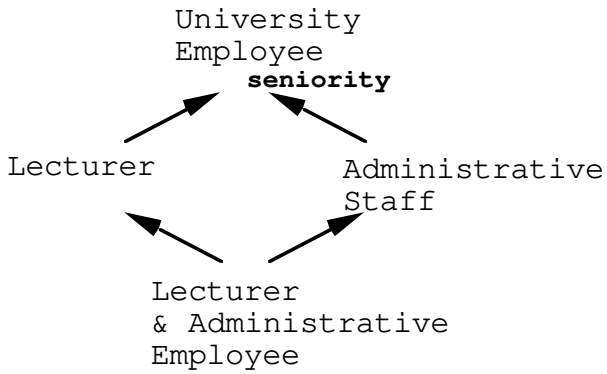

Figure 4.7

While designing the hierarchy we come to the conclusion that there are employees that take up both a position as lecturer and as administrative person. This is easily modelled by a class that multiply inherits from the lecturer class and the administrative staff class. What about seniority then ? The employee in question has two seniorities, one for each sort of employment. Of course this example only works fine with tree multiple inheritance. The transformation to a tree has the effect that for the "Lecturer&Administrative Employee" class, the Employee class is duplicated and as such also the seniority attribute.

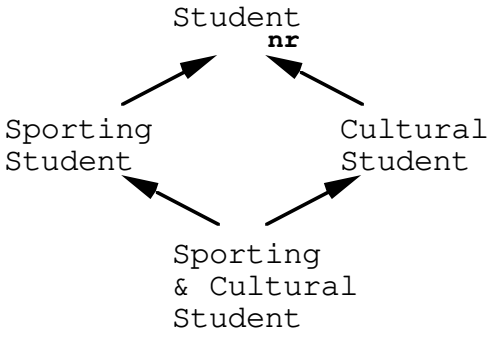

Figure 4.8

It is obvious that examples can be found where things don't work out as well as above. Consider the same university. This time we want to model a small part of the student database. Very similar to our employment database, there are two sorts of students: sporting students and students that take an interest in culture. Each student has a student number. The student number is used for administrative purposes. In a very sensible way, the management of the student number attribute is factored out in some abstract superclass student. For the example no other attributes are considered.

While designing the hierarchy we come to the conclusion that there are students that take up both an interest in sports as well as in culture. This is easily modelled by a class that multiple inherits from the sport student class and the cultural student class. What about student number then ? In contrast with the employee example all students have only one student number, regardless of

whether they are interested in sports or culture or both.

The student example can not be implemented easily in tree multiple inheritance. Not only because a cultural and sporting student will have two copies of his student number (that need to be synchronised), but also since the operations that are needed to manage this student number are now inherited from each duplicate of the student class and consequently raise name conflicts that need to be resolved. Solutions to this problem exist and are given in [Snyder87]. The idea is to avoid shared ancestors, leading to an extensive use of mixins, however.

The above problem imposes a serious limitation on the use of tree multiple inheritance since in most cases a very similar situation occurs if one wants to use a single root class in the inheritance graph that contains a set of general purpose attributes. These attributes are normally inherited by all objects in the system. With tree multiple inheritance, however, every class in the system that multiple inherits from two or more ancestors will need to resolve the conflicts that arise from the duplication of this root class. This imposes an absurd overhead on the use of multiple inheritance.

In conclusion, tree multiple inheritance avoids the exposure of inheritance that is inherent to graph multiple inheritance and implicitly linearised multiple inheritance, but does it at the cost of usability. Solutions to the problems of tree multiple inheritance exist, but must be found in an extensive use of mixins.

Point of View Notion of Multiple Inheritance

The *point of view notion of multiple inheritance* [Carré&Geib90] has grown out of a concern about the problems involved with *qualified message passing* as a means to resolve name conflicts. It handles somewhat orthogonal problems to the above discussed forms of multiple inheritance. Still we find it important enough to discuss it here.

Let's start with reviewing what possible problems can arise due to the use of qualified message passing. In the discussion qualified message passing means that every message (either an 'ordinary' message or an invocation of a parent operation) can be qualified with the name of a class. Any class that is in the inheritance chain of the receiver of a message can be used as qualifier. Due to this qualification method lookup starts from the specified class rather than directly from the class of the receiver of the message. It is important to note this since there are useful restrictions to the above scheme. One useful restriction is to allow only qualification for invocation of parent operations. This can further be restricted to allow only qualification with the names of the direct ancestors of the class that invokes the parent operation.

Unrestricted qualified message passing exposes the inheritance structure of each class. Not only does it make visible all the names of the ancestors of a class, it also allows a user of a class to select a non-most-specific definition of some attribute defined on that class (*refinement inhibition problem*). Furthermore, qualified message passing encodes too much information about the class hierarchy, which may change, as constant information [Bobrow&al.86]. Due to this it can disable further refinement of a certain attribute. Consider a class A, with two methods x and y. Although at first glance it does not seem so, there is a fundamental difference between sending a message y, from within the method x, to self, and sending a message y, from within the method x, to self qualified with class name A. Both have the same behaviour for instances of class A, but for inheritors of class A it is in the latter case impossible to refine (overwrite) the method y (*genericity inhibition problem*).

It would seem reasonable to impose the above mentioned restrictions on the use of qualified message passing. These restrictions can be interpreted such that from the viewpoint of some class the only information about the inheritance hierarchy that it can rely on is the list of its direct ancestors. It should also be noted that using a renaming technique for inherited attributes to solve name conflicts is as safe as this restricted form of qualified message passing. Renaming must be done explicitly by the programmer. This is an obvious disadvantage. On the other hand we will see later on in the text that renaming has an advantage over qualified message passing.

The effect of the above restrictions (and also for renaming as a technique for solving name conflicts) is that name conflicts must always be resolved in an explicitly defined inheritor. This is not always desirable. Consider the following example (example due to [Carré&Geib90]).

```
class SportsMan
 inherits Person
 extended with
    instance variables
       sportsmanNumber
    methods
      cardNumber:x
        if self validateCard:x
          then sportsmanNumber := x
      validateCard:x
        … check if x is a valid sportsman card number
      cardNumber
        ^sportsmanNumber
endclass

class Student
 inherits Person
 extended with
    instance variables
      studentNumber
    methods
      cardNumber:x
        if self validateCard:x
          then studentNumber := x
      validateCard:x
        … check if x is a valid student card number
      cardNumber
        ^studentNumber
endclass

class SportyStudent
 inherits SportsMan Student
endclass
```

Both the class SportsMan and Student are given classes. A possible combination of these two classes results in name conflicts for the methods that manipulate the card number of either the person as a student or the person as a sportsman. Still a student that is also a sportsman will have two different card numbers and as such a class that is a combination of the classes SportsMan and Student must respond to two sets of messages to manipulate the two different card numbers. Here, in this case some sort of qualified message passing, to differentiate between the messages sent to some person as a student and messages sent to that same person seen as a sportsman, seems appropriate. It must be possible to send messages to a person object seen from different viewpoints. Hence the name of this sort of multiple inheritance.

We will make a sharp distinction between the multiple viewpoint sort of inheritance, where the interfaces of the combined classes are kept separate, and all the other discussed sorts of inheritance where the name conflicts in the interfaces of the combined classes are explicitly resolved in the inheritor.

Although both sorts of inheritance have to address some of the same problems — e.g. the common ancestor duplication problem —, they both have to address problems that are specifically related to either sort of inheritance. The problem of duplicate invocation of parent operations is, for example, not relevant for the point of view sort of inheritance, because in the point of view approach conflicting methods are not combined in the inheritor.

One problem that is specific for the point of view sort of inheritance has to do with self reference in ancestor classes. This problem can be made apparent in the previous example. Take John a student that is also a sportsman. Let's presume that we can refer to e.g. the card number attribute of John as a student as '`John Student.cardnumber`' and to this same attribute of John as a sportsman as '`John SportsMan.cardnumber`' respectively. Irrespective of the fact that this sort of qualified message passing is problematic with respect to encapsulation (refinement and genericity inhibition problems), a simplistic approach to this sort of qualified message passing will fail with respect to self references in the so-invoked methods.

What will happen to the following valid message: `John SportsMan.cardnumber:423` ? The method lookup will correctly find the method named cardnumber: in the `SporsMan` class, and will invoke this method. This leads to the evaluation of '**self** `validateCard:423`', which is an unqualified message expression. The desired effect, of course, is that this expression is interpreted as '**self** `Sportsman.validateCard:423`'. Any other interpretation would lead to either an error or unpredictable behaviour. All 'naive' approaches to qualified message passing will fail to correctly interpret this sort of programs.

We will not give solutions to the above problem, for the time being it suffices to point out the problem. Solutions exist, in the form of a modified sort of qualified message passing [Carré&Geib90]). Later on in the text we will see that 'points of view' are strongly related to incremental modifications of objects in the prototype based approach to object-oriented programming.

One reason to introduce the viewpoint notion of multiple inheritance is that it enables us to give a crisp example of a particular sort of multiple inheritance that has largely been neglected, i.e. that of multiple inheriting of one and the same parent class. Consider again our sporting student example. Considering the fact that the `SportsMan` class and the `Student` class have very similar code it seems obvious to make the following abstraction:

```
class Member inherits Person
 extended with
  instance variables
    memberNumber
  methods
      cardNumber:x
        if self validateCard:x
          then memberNumber := x
      validateCard:x
        … check if x is a valid member card number
      cardNumber
        ^memberNumber
endclass
```

Of course this means that a possible sporting student class must inherit the Member class twice. The first time to express a student view and a second time to express a sporting view on the same person. It is obvious to see that, in the case that we do not want to define two 'empty' classes 'SportsMan' and 'Student' just to provide appropriate qualifiers, messages qualified with a class name are no longer sufficient in this example. Another means to make qualifiers must be provided.

Multiple Inheritance, Conclusions?
We confirm with Knudsen that:

> *"…by choosing strict and simple inheritance rules, one is excluding some particular usage of multiple inheritance …"*
>
> ***(Knudsen88)***

We add to this conclusion that there is a *trade-off between full encapsulation of inheritance and the expressiveness of the inheritance strategy* (in how effectively existing classes can be combined), and consequently, that it is sometimes necessary to expose the inheritance structure in a controlled way.

The trade-off between expressiveness and exposure of inheritance is apparent in two of the above examples. Firstly, in the example where one wants to avoid duplication of a shared parent (the sporting and cultural student example); secondly in the example where one wants to avoid duplicate invocation of a parent operation (the bounded history point example). In both cases it is necessary to expose some of the inheritance structure. In the former case both the '`SportyStudent`' class and the '`CulturalStudent`' class must expose the fact that they inherit from the 'Student' class and that they both don't mind that this parent will become a shared parent, so that the `Student` class can be shared in the '`Sporting&Cultural-Student`' class. In the latter case either the 'BoundedPoint' class or the '`HistoryPoint`' class must expose the fact that it inherits from the '`Point`' class and that it doesn't mind that another parent (hopefully with a similar behaviour) gets inserted between itself and its original Point parent, so that one of both can be assigned the other as parent (for the purpose of linearisation).

A possible solution could be devised where the programming language provides different inheritance operators: one that exposes inheritance and one that does not expose inheritance (much like in C++). Furthermore, a multiple inheritance operator that linearises the specified parents must be provided, one that keeps the inheritance graph as is and one that duplicates specified parents. This is more or less the direction taken in [Knudsen88], although there, the set of inheritance operators has been restricted to those that control the duplication and sharing of shared ancestors (unification inheritance and intersection inheritance).

In this text we propose a different solution based on mixins. We will extend the mixin-based approach with a mechanism to resolve name conflicts and we will show that, given this extension, mixins are sufficient to express all the above multiple inheritance hierarchies in an effective and simple way. Mixin-based inheritance was not only chosen because of its capacity to effectively construct multiple inheritance hierarchies but also for its capabilities to control and abstract over how these hierarchies are constructed, given the fact that mixins can be seen as attributes. The scope rules that emerge from the use of nested mixins also play an important role. We will show that mixins are exactly the right building blocks to construct (multiple) inheritance hierarchies.

4.2.4 Mixin-based inheritance

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

A mixin-class in CLOS is a class that has no fixed superclass and as such can be applied to different superclasses. In CLOS terminology, this means that a mixin-class can invoke a Call-Next-Method, even though it has no apparent superclass. Mixin-classes in CLOS depend directly on multiple inheritance, and more specifically linearisation, for them to work.

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

Mixin-Based Inheritance
Contrary to mixin-classes, in *mixin-based inheritance*, a mixin is not a class (a mixin cannot be instantiated for example), and multiple inheritance is a consequence of, rather than the supporting mechanism for, the use of mixins. In contrast to CLOS, in which mixins are nothing but a special use of multiple inheritance, mixins are promoted as the only abstraction mechanism for building the inheritance hierarchy [Bracha&Cook90] [Bracha92] [Hense92] [Codenie,Steyaert,Lucas92] [Steyaert&al.93].

To introduce mixins, we must return to our model of inheritance of the previous chapter. Inheritance was modelled as an incremental modification mechanism where a parent P (the superclass) is transformed with a modifier M to form a result $R = P \Delta M = P + M(P)$.

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. In [Bracha&Cook90] it is also shown that mixin-based inheritance subsumes the inheritance mechanisms provided in Smalltalk, Beta and CLOS.

Whereas in conventional single or multiple inheritance the modifier M has no existence of its own (generally it is more or less part of the result R), the essence of mixin-based inheritance is exactly to view the modifier M as an abstraction that exists apart from parent and result. Modifiers are called "mixins". The composition operation Δ is called "mixin application". The class to which a mixin is applied is called the base class. In "pure" mixin-based inheritance, classes can only be extended through application of mixins.

The Δ operator sees to it that the parent P is passed as explicit parameter to the modifier M. In practice a mixin does not have its base class as explicit parameter, but rather, a mixin has access to the base class through a pseudo variable, in the same way that a subclass has access to a superclass through a pseudo variable

(e.g. the "super" variable in Smalltalk). In a statically typed language, though, this means that a mixin must specify the names and associated types of the attributes a possible base class must provide. This is why mixins are sometimes called "abstract subclasses".

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

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

mixin-based inheritance
M is mixin
  defining NamedAttribute1 ... NamedAttributen
  applicable to base-class with[2]
    SuperAttributeSignature1 ... SuperAttributeSignaturen
  endmixin
class R1 inherits P1 extended with M endclass
class R2 inherits P2 extended with M endclass
```

4.2.5 Mixin-Method Based Inheritance

Mixin-based inheritance in the above form is an inheritance mechanism that is directly based on the model of inheritance as an incremental modification mechanism. It makes wrappers and wrapper application explicit [Bracha&Cook90] [Hense92]. In this section we generalise mixin-based inheritance in three ways.

Our mixins are based on a more general form of wrappers, where wrappers can have multiple parents. The notion of wrappers with multiple parents has already been pointed out in [Cook89]. The notion of multiple parents will be used to solve name-collision problems for multiple inheritance hierarchies where the interfaces are merged.

Furthermore, we extend the use of mixins to object-based inheritance. This sort of object-based inheritance is similar to implicit anticipated delegation [Stein,Lieberman&Ungar89], the resulting objects are comparable to split objects of [Dony,Malenfant&Cointe92]. We will show how this solves the problem of name-collisions in multiple inheritance hierarchies where the interfaces are not merged.

And finally we address the question of how mixins can be seen as named attributes of objects in the same way that objects and methods are seen as named attributes of classes. The general idea is to let an object 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.

---

2 This specification will be omitted in further examples for reasons of brevity.

Mixins with Multiple Parents

Consider the following classical example for multiple inheritance: we have a class `Car` and a class `Toy` and we want to combine their features to make toy cars. We want to merge both interfaces, so that only one version of the `print`-message is applicable to `ToyCar` (for the example no other attributes will be considered). In the definition of the `print`-method on the `ToyCar`-class we want to invoke the `print`-methods of both parents. That way we can combine e.g. the brand name of the car and the size of the toy to form the specification of the toy car. To do this, we need a mechanism to combine the conflicting methods.

```
Car-Mixin is mixin
  defining
    instance variables brand
    methods
      print
        brand print
 endmixin

Toy-Mixin is mixin
   defining
    instance variables size
    methods
      print
        size print
 endmixin

ToyCar-Mixin is mixin
  defining
     methods
      print
        -- invoke print operation of my 'car' part
        -- invoke print operation of my 'toy' part
  endmixin
```

Simple qualified message passing does not work for mixin-based inheritance since a mixin does not have a single base class that could serve as a qualifier. In the above example the `ToyCar-Mixin` mixin has no knowledge of the base classes to which it will be applied. In fact, it does not even know whether the car or the toy mixins will be part of the base class to which it will be applied.

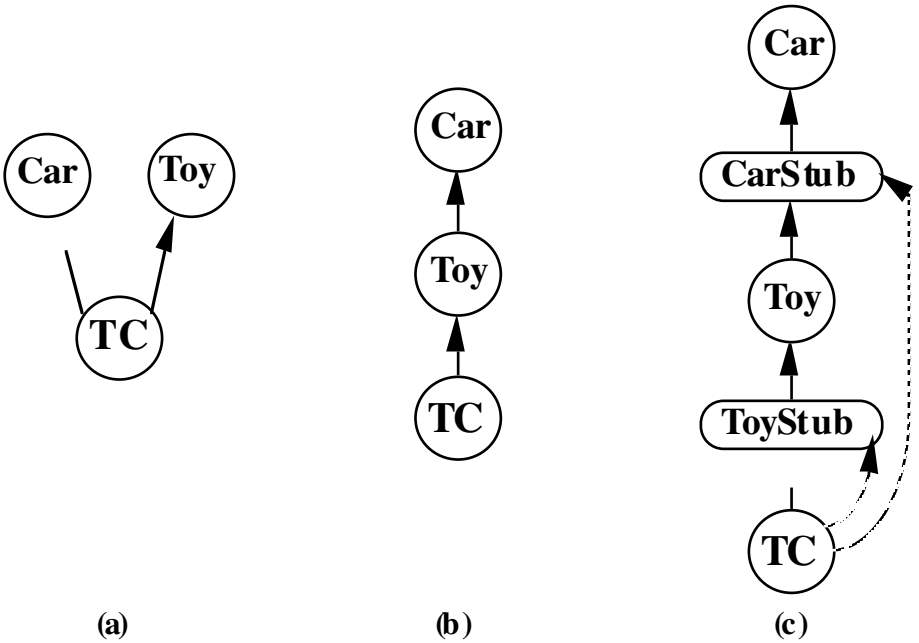

Figure 4.9

The problem here is that we have to deal with a linearised inheritance chain (figure 4.9b), but we still want to be able to refer to non-direct super classes (i.e. we want to simulate figure 4.9a). To do this we have to bring some 'hierarchy' into the chain. We already mentioned in the introduction that the solution should be found in mixins that have access to non-direct parents. We therefore introduced the notion of *stubs*. Just as mixins, the stubs have to be inserted at the right place in the inheritance chain (figure 4.9c). In this manner subclasses can use non-direct superclasses as parameters and 'mimic' a graph structure in the linear chain (dashed arrows in figure 4.9c). Stubs then serve as pointers to the place in the inheritance chain where method lookup should start when invoking parent operations.

```
ToyCar-Mixin is mixin
  needs Car-Stub Toy-Stub
  defining
     methods
      print
        print super:Car-Stub
        print super:Toy-Stub
  endmixin

class ToyCar inherits Root
  extended with Car-Mixin defining Car-Stub for ToyCar-Mixin
  extended with Toy-Mixin defining Toy-Stub for ToyCar-Mixin
  extended with ToyCar-Mixin
endclass
```

Using these stubs, a `ToyCar`-mixin can be created, that solves the name conflicts appearing when combining `Toy` and `Car`. To avoid problems with self references in inherited methods, *all* name conflicts have to be explicitly resolved here. It is not sufficient to simply resolve name conflicts occurring through combination of `Car` and `Toy`. Consider different implementations of the `print`-methods in the `Car`- and `Toy`-mixins, that do a self send of e.g. a message `getName`. This method could equally well be implemented in one of the ancestors of `Car` or `Toy`. It is therefore necessary to resolve *all* name conflicts in `ToyCar`.

The use of stubs must be restricted so that they can only be used to invoke parent operations of non-direct parents. In the case of the toy car, only the `ToyCar`-mixin should be able to use `Car-Stub` and `Toy-Stub`. On the other hand the definition of stubs cannot put constraints on the order in which mixins are applied. A concrete realisation of stubs should respect these constraints.

Separated Interfaces

Let us return to the example used in the introduction of the point of view notion on inheritance. We clearly want to keep the interfaces of the `Student` and `Sportsman` classes strictly separate. We want to be able to treat a `SportyStudent` as a student or as a sportsman, depending on the situation. We already mentioned that points of view are strongly related to incremental modifications of objects in the prototype-based approach to object-oriented programming.

In the previous discussion we left implicit the fact that mixins can be applied to objects. However, mixins can be used to dynamically extend objects in a prototype-based approach to object-oriented programming. New objects can be created by taking an existing object and extending it with a set of variables and methods. Similar to mixins in a class-based language we can identify a base object and a set of extensions. Here as well, extensions can be considered as separate abstractions. The terminology mixins and mixin application from the class-based case can be retained. Application of mixins to objects is an important part of the solution to our multiple inheritance problems. Consider the following example.

```
Person is class ...

Sportsman-Mixin is mixin
   — same definitions as before
 endmixin

Student-Mixin is mixin
   — same definitions as before
 endmixin

john is instance of Person;
...
johnAsASportsman is john extended with Sportsman-Mixin;
johnAsAStudent is john extended with Student-Mixin;
```

In the code displayed above, we first create an instance `john` of class `Person`. We can then create two new objects, `johnAsASportsman` and `johnAsAStudent`, each representing a different view on `john`. Being two dynamic extensions of `john`, they share its attributes (i.e. the attributes of `Person`).

Furthermore, as we now send messages to these new objects, the self reference problem is also resolved. When `setCardNr` is sent to either `johnAsASportman` or `johnAsAStudent`, `self getCardNr` is accordingly sent to this same initial receiver object. All `Person`-messages sent to `johnAsASportman` or `johnAsAStudent` are implicitly delegated to `john`.

Mixins as Attributes: Mixin-Methods

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

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 amount into a rather static inheritance hierarchy and duplication of mixin code (each mixin would be applicable to only one object).

A mixin can be made applicable to more or less objects according to its position in the inheritance tree. The higher it is defined, the more objects that can be extended with this mixin. In a programming language (such as Agora) where mixin-based inheritance is the only inheritance-mechanism available, this means that all generally applicable mixins (such as a mixin that adds colour attributes) must be defined in some given root object.

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

CarMixin is mixin
  defining enginetype
endmixin

car is object obtained by CarMixin extension of Root
—— car inherits ColourMixin from the Root object
colouredCar is object obtained by ColourMixin extension of Car
```

4.2.6 Mixin-based inheritance, A Solution to Multiple Inheritance Problems ?

Applicability of Mixins
An object lists as mixin-attributes those mixins that are applicable to it. What defines applicability of a mixin to an object ? 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]. In a regime where nothing but behavioural compatible modifications are allowed, only the mixins that define a behaviour compatible modification of a certain object are applicable to that object.

To put it another way, restricting the applicability of mixins puts a constraint on the possible inheritance hierarchies that can be constructed. This could answer our desire to constrain multiple inheritance hierarchies.

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 both 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 realised simply by cancelling the `Male-Mixin` in the `Female-Mixin`, and by cancelling the `Female-Mixin` in the `Male-Mixin`. This solution relies on the ability to cancel inherited attributes. Other more formal solutions can be developed.

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

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

A Global View on the Inheritance Graph with Mixins
Mixin-based inheritance causes *explicitly linearised inheritance*. The order in which mixins are applied is important for the external visibility of public attribute names. Attributes in the mixin override the attributes of the base class having the same name. In absence of any name clash resolution mechanism, attribute name lookup is determined by application order.

Apart from this explicit linearisation, 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.

Since mixin-based inheritance gives rise to linearised inheritance, it is obvious that the undesired duplicate invocation of parent operations can be resolved with mixins. A possible implementation of the bounded history point can be found below.

```
Point-Mixin is mixin
  defining
    instance variables x  y
    methods
    moveX:dx moveY:dy
      x := x + dx
      y := y + dy
 endmixin

History-Mixin is mixin
   defining
    instance variables history
    methods
    moveX:dx moveY:dy
      history record: ("moved to: ", self location)
      super moveX:dx moveY:dy
 endmixin

Bounds-Mixin is mixin
   defining
    instance variables bounds
    methods
    moveX:dx moveY:dy
      if self (location getX + dX) within: bounds
           & (location getY + dY) within: bounds
      then super moveX:dx moveY:dy
 endmixin

class BoundedHistoryPoint inherits Root
  extended with Point-Mixin
  extended with Bounds-Mixin
  extended with History-Mixin
endclass
```

As the programmer has total control over the linearisation, there are no unforeseen insertions of unrelated classes between a class and its parent. This leads to the preservation of encapsulation of the inheritance hierarchy and makes parent invocations safe, as one always has control over the direct parent. It should be noted that the above solution is heavily based on global information of the inheritance graph. The bounded history point can only be constructed as a linear chain of the point, history and bounds mixin because we have information about the way each of them invokes parent operations. Since a mixin is an abstract subclass, the parent operations it invokes are part of its interface.

Let us now take a look at the common ancestor duplication problem. Reconsider

the examples of the sporty student and the univesity employee that were used to illustrate the common ancestor duplication problem. It is obvious that these are examples of the view on inheritance in which the interfaces of the combined classes are kept strictly separate. They can easily be described in a mixin-based approach. One remaining problem is that we want duplication of the attributes inherited from the common ancestor in the first example, while we want only one shared copy in the latter. This can easily be resolved here by applying an `Employee-Mixin` twice in the first example, and a `Person-Mixin` only once in the second. This results in the following definitions.

```
john is instance of Person;

johnAsASportsman is john extended with Sportsman-Mixin ;
johnAsAStudent is john extended with Student-Mixin ;

johnAsALect is john extended with Employee-Mixin Lect-Mixin
johnAsAnAdmin is john extended with Employee-Mixin Admin-Mixin
```

Mixin-Methods, Conclusions and Open Questions

In response to our analysis of multiple inheritance we proposed an inheritance mechanism based on mixins. We extended the mixin-based approach with mechanisms to resolve name conflicts. We showed that, given these extension, mixins are sufficient to express an entire range of multiple inheritance hierarchies in an effective and simple way. Mixins are so expressive because they allow unanticipated combinations of behaviour to be made. If uncontrolled, one faces an explosion of possible combinations of mixins. A mechanism to control this combinatorial explosion is needed. Mixin-methods are proposed as a uniform framework to control and make abstraction of the way multiple 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 approach suggested was based on two different views on the inheritance hierarchy. In one view the interfaces of the combined classes were merged, in the other they were kept separate. One question now suggests itself: should it be possible to combine these two views ? In other words, should we be able to merge and separate interfaces within one single branch of the hierarchy ? These problems are related to the problems with split objects [Dony,Malenfant&Cointe92] and to the modelling of inheritance with explicit bindings [Hauck93]. They are left open for future research.

Another question that was left open is a more formal approach to restricting the applicability of mixins. In the next section we will show how mixin applicability can be restricted when a mixin depends on the implementation details of the base class it is applied to. A natural form of nesting of mixins will result from this. Other mechanisms for restricting the applicability of mixins will be shown in the section on extensions to Agora.

4.3 Visibility and Nesting in Object-Oriented Languages

Visibility rules are an important issue in the design of a programming language. In an object-oriented language this is especially so. Names (Identifiers) are very important in object-oriented languages. Objects are essentially collections of *named* attributes and defining an object is essentially the definition of a collection of names. Furthermore, the notion of encapsulation puts an a priori restriction on visibility.

This section is not about visibility rules for attributes that are in the interface of some object and are accessible through passing a message to that object (e.g. this section is not about different views on one object), but rather about the visibility rules of the attributes that are directly (i.e. without message passing to an explicit receiver) accessible for some object, and that are part of this object's encapsulated part.

The scope of an identifier is defined as the program code in which this identifier is visible. A name space is defined as a collection of all identifiers with a same given scope. Name spaces can be nested.

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, but rather for each kind of "variable" a different lookup strategy is defined. Smalltalk for example, has a blend of variables (class variables, class instance variables, global variables, pool variables, instance variables, arguments, local variables) each with their own visibility rules.

Block 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[3]. Block structures are a natural way to hierarchically structure name spaces (modules are an alternative). Accordingly scope rules can be imposed. Typically the scope of an identifier declared in some block includes this block and all the blocks enclosed in it, but not the enclosing blocks.

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.

One must take care 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 subobject[4] within an object has a separate name space. The intention is to regulate the sharing of name spaces of subobjects. While this breaks the encapsulation of subobjects, 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 subobject can access the encapsulated part of another subobject within the same object (mediated by the above discussed rules of course).

---

3 In Smalltalk this is partially remedied with the category concept. Classes are organised into categories. Categories however are only for documentation purposes.

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

Sometimes there is a need to share name spaces between objects, rather than subobjects. The above mentioned class variables and global variables, as found in Smalltalk, are examples of such name spaces that are 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 subobjects, it is obvious that another set of scope rules can regulate the creation of shared name spaces for objects. This is normally what is done with nested classes, and will be discussed in the section on class nesting.

4.3.1 Is There a Need for Scope Rules for Encapsulated Attributes ?

The visibility rules that are discussed in this section apply to the names of the private attributes of an object. Alternatively, visibility rules can be, and have been, devised for the names of the public attributes. In the normal case any object can send any message to all the objects it has knowledge of, provided that the messages it sends are in the receiver's interface. Sometimes it is desirable to restrict this. In its most general form the invocation of a particular method can be restricted to a limited set of objects. Vice versa, it is possible that all objects that have knowledge of some common object, can not all send the same messages to this common acquaintance. In most cases visibility restrictions on the interface of an object take a particular form. Visibility is, in these cases, based on the fact whether one object is derived from another object, according to inheritance.

It can be argued that in presence of visibility restrictions on the interface of an object, there is no need for encapsulated attributes. In fact this is argued in [Ungar&Smith87] for the programming language Self. We will take a closer look at this.

Self is a slot-based language, i.e. variable access and method invocation take the same form. This is realised by access-methods. Each variable declaration introduces two access-methods: one to retrieve the value and one to store some value in the variable. Hence no assignment statement nor identifier lookup is needed in slot-based languages. All variable access takes place by sending messages.

Uniform access to state and behaviour has certain important advantages. First of all the message passing paradigm is not diluted with assignment and state access or identifier lookup. Secondly it allows a programmer to freely re-implement variable declarations into corresponding method declarations without having to check all the users of a variable. Furthermore, when inheriting from a class, all variable declarations can be overridden with method declarations (override both or one of both access-methods), or vice versa.

Since in slot based languages all variable access is done by message passing, this leads to very verbose programs. An object that needs to access one of its variables must send a message to its 'self', the receiving object. Special provisions are made to overcome this problem. For instance, in the language Self all messages that are sent from an object to this object itself (i.e. to the self) can omit the receiver. In the remainder of this text messages with an implicitly determined receiver (not necessarily the self), will be called 'receiverless messages'.

Self objects are 'flat' objects. All attributes (called slots) are part of an object's interface, including those slots that correspond to what would be called instance variables elsewhere. Encapsulation is realised by dividing the interface in public slots and private slots. All possible objects can send messages that invoke public slots. Only an object itself can send messages that invoke private slots. As such Self could be called an object-oriented programming language with encapsulation. Encapsulation depends on the identification of the sender of a

message. An object has, as a sender of messages to itself, a greater accessibility to itself than other objects. However, we will argue that the encapsulation achieved this way is essentially module based.

Of course, visibility rules for interfaces of objects interfere with inheritance. Are visibility properties, associated to some method, inherited or not ? This question is the more relevant, in view of the encapsulation of inheritance, if these same rules are used to enforce encapsulation. We will explore some variations for public and private slots, and we will show that only module based encapsulation can be supported.

A private slot can not only be made visible for the container object itself but also for its inheritors. This corresponds to non-encapsulated inheritance: an inheritor can access the private slots of its parent. It should be noted that an object can also access the private slots of non-direct ancestors. This need not always be desirable.

Conversely, a private slot can be made visible to the ancestors of the container object. This is not only a prerequisite for exploiting the full potential (i.e. the ability to override variable slots with methods in inheritors) of slot-based languages, but results in a cumbersome semantics otherwise. As depicted in the following figure, an ancestor can only access its own private attributes by sending messages to the inheritor that it is part of. Consequently the ancestor also has access to the private slots of its inheritor.

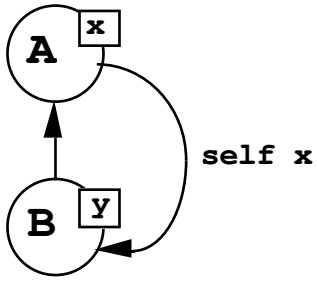

Figure 4.10

Private slots that are visible to ancestors can give rise to *module based encapsulation* [Ungar&Smith87]. In prototype based languages parent-objects can be used to store shared attributes. Typically, the collection of methods that implement a data type are stored in an object shared by all members of that data type. Parent objects that are used for this purpose are called 'traits' objects in Self. Traits objects play the role of classes in prototype based languages. The result is that all objects that belong to a certain 'data type' (e.g. all point objects) are derived from one and the same traits object. The traits object has as ancestor access to private slots of all the elements of the data type it implements. Every method in the implementation of the data type has access to the private slots of the elements of the data type.

Even in cases where private slots are made visible only to an object itself (and not to its descendants or ancestors), it can be shown that this leads to a degenerate form of module based, rather than object-based encapsulation. This can be observed in a method that takes an actual argument that is the same as the receiver of the method. This method has, in contrast with object-based encapsulation, access to all the private slots of both the receiver and the argument object.

As we saw earlier module based encapsulation is useful in cases where for example two elements of one and the same data type need to be compared. Other

useful examples include the initialisation/creation of elements of a data type. However, as we argued earlier module based encapsulation is best provided by a separate language construction, i.e. modules.

To conclude: although visibility restrictions on the interface of an object are very useful and can be used as a way to achieve encapsulation, they serve the purpose of encapsulation far from perfect. The essence of the above is that visibility restrictions on interfaces allows the separation of private from public attributes, but nothing is provided to structure the name space of the private attributes. We will see in the following sections that there is a need to structure the private name spaces of objects that are more sophisticated. Two important concepts were discussed in this section: that of slot based languages, i.e. languages where message passing is not diluted with state access primitives, and that of receiverless messages, i.e. messages that have an implicitly determined receiver.

4.3.2 Nested Classes, Classes as Attributes

Having classes as attributes, or having *nested classes* can serve two different purposes:

- locally visible classes: classes that are only visible in some local context
- locally defined classes: classes that are only meaningful in some context

In most object-oriented languages, classes reside in a name space shared by all objects. Indeed, almost all classes should be visible for all objects. Still, the ability to have classes that can only be used in a limited context is useful. In most cases this amounts to having a class as attribute-value of a private attribute of some object or class. On the other hand for class-based inheritance the ability to have classes as attributes normally implies class nesting, and as such it is not possible to have locally visible classes without nesting.

Apart from restricting the scope from some class, a class can be nested in another class in order to express the fact that a class only has meaning in relation to an instance from the enclosing class. This kind of nesting usually occurs when the nested class is a public attribute of instances of the enclosing class. Consider the following example (example from [Madsen&Møller-Pedersen89]):

```
class Grammar
  class Symbol
    methods
      isTerminal
      isNonTerminal
  end Symbol ;

  methods
    ... grammar methods come here
end Grammar ;

G1 : Grammar;                S1,S2 : G1.Symbol ;
G2 : Grammar;                T1,T2 : G2.Symbol ;
```

A class Grammar is defined and a class Symbol is defined local to this Grammar class. Notice that there is no sub- or superclass relation between Symbol and Grammar. The example is such that the class Symbol is a public attribute of instances of class Grammar. The class Symbol is used to represent the lexical symbols that occur in a grammar. With each grammar a different set of symbols is used. This is naturally expressed by the fact that the class Symbol is an attribute of instances of — and is nested in — the class Grammar. As can be seen in the

example, instances of the Symbol class can only be created by selecting the Symbol attribute of an instance of grammar. In fact we can talk about different 'versions' of class Symbol. In the example the classes `G1.Symbol` and `G2.Symbol` are different, and in some sense incompatible, classes.

As said before, for class-based inheritance the ability to have classes as attributes normally implies class nesting. When nesting classes, the scope rules that are naturally connected with nested block structures result in shared name spaces for objects. This is obviously apparent in the grammar example. Presume the existence of a set of instance variables in the class grammar. Instances of some version of the Symbol class have direct access to the instance variables of the grammar instance on which they depend. Although the example is taken from BETA, a language in which objects are not encapsulated, we will illustrate the effect of nested classes on encapsulation.

Although nested classes can be very useful (as is shown in both [Madsen87] and [Buhr&Zarnke88]), they 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. Modification of this variable in, let's say b1, has an effect on the variable seen by a and b2. So, instances of class B can directly access the instance variables of an instance of class A even if there is no sub or superclass relation between A and B.

```
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
```

Non-encapsulated inheritance and nested classes do not mix very well. This is apparent in languages such as BETA [Madsen87]. Identifier lookup 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 super class of B. This problem is resolved by giving priority to one of both name spaces in case of a name clash, e.g. by first looking in the super class chain and then in the surrounding scope (here again the superclass chain must be searched and so on …). Ironically, nested classes, that can be used to break the encapsulation of objects, can only be used unambiguously in a language with strictly encapsulated inheritance.

4.3.3 Nested Mixins

In most object-oriented languages a subclass can access its superclass in two ways. One, by direct access to the private attributes of the superclass (direct access to the implementation details). Two, 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 trade-off 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 done 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, recognising the fact that in some cases direct access to a base class's implementation details is needed. To put it otherwise: 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. Essentially mixins are differentiated by how much of the implementation details of the base class are visible to them.

```
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, but the visibility of C
attributes is different for both subclasses ——
```

The degree to which a mixin has access to the implementation details of a class is solely based on whether a mixin is a proper or an inherited attribute of a certain class. Consider a class C that was constructed by application of a mixin MC to some base class. There are two sorts of mixins that can be used to create subclasses of C: mixins that are proper attributes of C (in the example: PMC defined in the mixin MC) and mixins that are inherited (in the example: NotPMC). 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. A mixin that is inherited has no access to the proper private attributes of the class it is applied to. Note that this leads in a natural way 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, and according to lexical scope rules has access to the names of the attributes defined in the mixin MC.

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 the instance variables defined in a mixin 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, resulting in a totally encapsulated form of inheritance, as is proposed in [Snyder87]. It is called abstract because the resulting subclass must use message passing to access inherited private attributes.

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. Consequently it is very easy to construct Smalltalk-like inheritance using this approach.

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. Thanks to the fact that the applicability of a mixin is restricted to the classes where it is defined on, it is always guaranteed that the resulting object is consistent in the sense that components referred to through nesting always exist.

It is obvious that the possibility to nest mixins provides the user with a very powerful tool for building inheritance hierarchies. Instead of promoting one single strategy for handling encapsulation between inheriting clients (instance variables are either always or never visible in subclasses) one allows the user to build his own application specific encapsulation mechanism.

4.4 Design of Agora

4.4.1 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 is innovative in the way that these features are bound together in one consistent language framework. 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. Whereas in most programming languages, e.g. inheritance and name-binding mechanisms are expressed in structures that differ fundamentally from ordinary programming structures, Agora requires but one

programming paradigm for all components of the system.

4.4.2 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). For clarity, in the text 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 *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 private methods; they will also be used as part of other syntactic structures where message patterns need to be manipulated (i.e. method declarations).

```
size                        receiverless unary message (identifier)
+ aString2                  receiverless operator message
at:index put:aChar          receiverless keyword message
```

A third category of message expressions is the category of *reify messages*[5]. Reify messages have the same syntax as message expressions, and respectively receiverless message expressions except for their bold-styled 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 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.

```
self                        receiverless unary reifier
a <> 3                      operator message reifier
a define: 3                 keyword message reifier
```

Message expressions can be grouped to form blocks. Blocks are an example of the third kind of reify expressions, i.e. reify expression whereby 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.

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

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

The following shows the concrete grammar of Agora 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

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    -> "]" | "}"
```

4.4.3 Standard Agora Reifiers

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 section on reflection we will show how the set of reifiers can be extended. For the time being we will need a standard set of reifiers. We will not try to be complete in this list. The idea is to define a vanilla variant of Agora, that can be used in a subsequent section to be extended with more elaborate constructions.

Variable Slots
Variables, be it instance variables or local variables, are declared with a variant of the **define** reifier. Its three standard variants are listed below.

```
x define                    variable declaration reifier
x define: 3                 same, but with initial value
d <> Dictionary             same, but with clone of initial value
```

Agora is a slot based language. The value of its variables must be accessed and modified through message passing. The receiver, however, can, in case of an access to a private variable, be left implicit. So, a private 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 also:

```
x                           variable access
x:3                         variable assignment
x <- 3                      assignment reifier; equivalent to the above
```

A note should be made here. Since receiverless messages are used to access variables, they must, in contrast with other messages, obey the lexical scoping when being looked up (more on this in the section on nested mixin methods).

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 'blocks' (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 former will be discussed as an extension to Agora in a later section. For the time being, examples of the standard control structure reifiers are listed below.

```
a = b ifTrue:[a <- 3 ]
      ifFalse:[a <- 4 ]                 if reifier

a < b whileTrue:[a <- a + 1 ]           while reifier
```

More importantly Agora has a self pseudo variable reifier and a reifier to invoke parent operations. Both reifiers are receiverless.

```
self                        self pseudo variable reifier

super:(at:3 put:5)          super message invocation reifier; invokes the
                            parent's ' at:put:' method
```

The self pseudo variable reifier, when evaluated, returns the current receiver. The super reifier delegates its message argument (which must be receiverless) to the parent object. The super reifier has a bit of an 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 private operations (and in this variant of Agora only identifier lookup is supported), the latter to invoke public operations. So '**self** x' and 'x' generally have a different result.

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 object called `Object`. So, in the example below, the root object `Object` is extended with colour attributes by invoking its *addColour* mixin (sending the message *addColour* to it). The resulting `ColourObject` object is an inheritor of `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, for example 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.

Although we often used the term 'class' in the explanations of the above examples, Agora is at its heart an object-based language with object-based inheritance. In fact in 'standard' Agora no notion of classes exists. Agora's mixin method are applicable to objects. An object is extended by invoking one of its mixin methods.

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 activation 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 more permanently. Due to this similarity, arguments and local variables of methods can be defined and accessed in a totally similar way as 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 !

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 in a later section. 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.

```
p1 <- (P clone)     assign a copy of P to p1

p1 <> P             declare a new variable p1 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.

In the following example two extensions of a same, shared parent (john) are

made. These extensions implement different views on the same object, in this case to resolve the 'card number' name conflict. This example encodes, in Agora, the 'multiple viewpoint' example from the section on multiple inheritance.

```
MakePerson Mixin:
  [ name define ;
    name:newName Method:[name <- newName] ;
    name Method:name ] ;

MakeSportsman Mixin:
  [ cardnumber define ;
    number:newNr Method:[cardnumber <- newNr ] ;
    number Method:cardnumber ] ;

MakeStudent Mixin:
  [ cardnumber define ;
    number:newNr Method:[cardnumber <- newNr ] ;
    number Method:number ] ;

Person define: Object MakePerson ;

john <> Person ;
johnAsASportsman define ;
johnAsAStudent define ;

johnAsASportsman <- john MakeSportsman ;
johnAsAStudent <- john MakeStudent ;

john name:'john' ;
johnAsASportsman name                    ---> 'john'
johnAsAStudent name                      ---> 'john'

johnAsASportsman number:4 ;
johnAsAStudent number:5 ;
johnAsASportsman number                  ---> 4
johnAsAStudent number                    ---> 5
```

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 an inheritor depends on the implementation of its heir. 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.

```
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:
            [ super:home ; bound <- Circle m:location r:infinite ] ;
        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)] ] ] ] ;
Turtle define: Object MakeTurtle ;
BoundedTurtle define: Turtle MakeBounded ;
aBoundedTurtle define: BoundedTurtle clone ;
aTurtle define: Turtle clone ;
aBoundedTurtle forward:1 ;
aTurtle forward:3
```

Once the turtle is defined, the next step is to create an inheritor 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 inheritor 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 prototypes `Turtle` and `BoundedTurtle`, two mixins, `MakeTurtle` and `MakeBounded` respectively, are defined. To make sure that the prototype `BoundedTurtle` inherits from prototype `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` prototype and its inheritors. Not only is it impossible to extend the root object `Object` with the `MakeBounded` mixin since it is not defined for the root object but also since `Object` does not define the `location/heading` instance variables that are required by the `MakeBounded` mixin.

Each clone of `Turtle` and each clone 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 an inheritor (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, 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.

```
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` object 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 an accessor method to the location of the turtle must now be made public, 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 objects that respect the `forward`/`position` protocol.

Earlier on we said that the `MakeBounded` mixin could only be applied to the `Turtle` object and its inheritors. `DrawingTurtle` is such an inheritor. 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. 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 object 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` object'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*. Not all mixins are commutatively applicable. Therefore the order of mixin application must not have an effect on the exposure of implementation details of the applied mixins to each other.

Mixin Methods
The fact that mixin application is realised 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, and an example of late binding of mixins.

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 given making use of the Turtle objects 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 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.

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

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

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

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

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

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

4.5 The Agora Framework

The experiences, techniques and terminology acquired in building the framework for Simple can now be put to use in building a framework for a full-fledged object-oriented programming language. The standard variant of Agora is used for this purpose.

We will see that the kernel of the framework — i.e. expression, object, pattern and slot classes —, is shared between Simple and Agora. This should come as no surprise. The differences between Agora and Simple are important however. In Simple the encapsulation operator played an important role. In Agora this operator is abandoned[6,7], and a more standard way of argument passing is employed. Simple has no inheritance, inheritance is an important aspect of Agora. Agora is entirely built around generic expressions, called reify expressions. So, Agora is not a specialisation of Simple, rather Simple and Agora are both specialisations of the same framework.

An important aspect of Agora is how much the representation of objects is affected by the inheritance structure. Especially for object-based inheritance, one could be tempted to encode the inheritance structure by means of delegation. An example is given below.

```
class ShouldBeDelegatedToObject
  methods
    abstract delegate:pattern receiver:receiver
      result ShouldBeDelegatedToObject
endclass

class ObjectWithParent extends ShouldBeDelegatedToObject
  instance variables
    slots: Set(Slot)
    parent: ShouldBeDelegatedToObject
  methods
    concrete delegate:pattern receiver:receiver
        result ShouldBeDelegatedToObject
      slot := slots findSlot:pattern
      if slot found
         then [... evaluate the body part of the slot ... ]
         else [^parent delegate:pattern receiver:receiver]
endclass
```

6 Incorporating this encapsulation operator in Agora is a non-trivial task due to interference with Agora's inheritance. We think, however that it is an important lack in Agora since this issue is related to the issue of virtual private attributes (that are also lacking in Agora).

7 Note that this does not mean that encapsulation is abandoned. Agora's objects still make a distinction between encapsulated and public attributes.

Here objects (i.e. objects of class `ObjectWithParent`) that have a parent object, will, upon reception of a message, look up the corresponding slot in their own set of slots and delegate the message if the right slot is not found. For this to work, all objects must be represented as instances of concrete subclasses of the abstract `ShouldBeDelegatedToObject` class, that specifies the nature of delegation.

Such an implementation can be discarded as being too operational. In our analysis of object-oriented programming languages, we discarded those languages that have an explicit delegation operator. So, when introducing an explicit delegation operation in the implementation of objects, again, at the implementation level we have a finer view to distinguish objects than is possible at the programming language level. Thus, such an implementation is not 'fully abstract'. We will show that it *is* possible, even in the presence of inheritance, to maintain our abstract representation of objects.

4.5.1 Abstract Grammar, Expression Objects and Reifier Methods

Agora takes the notion of generic expressions to the extreme. Its syntax (see previous section) is built up solely with message passing expressions and generic expressions. Generic expressions in their own right are formulated in the form of message expressions. Message passing forms the kernel of Agora.

Agora Abstract Grammar

```
Non-Terminal          = { Aggregate, Message, ReifierMessage, UnaryPattern,
                          OperatorPattern, KeywordPattern, ReifierUnaryPattern,
                          ReifierOperatorPattern, ReifierKeywordPattern }

Terminal              = { Identifier, Operator, Keyword, Literal, Delimiter }

Root                  = ExpressionSet

—— expansion sets ——
ExpressionSet       = { Literal } + { Aggregate }
                      + { Message } + { ReifierMessage }
                      + PatternSet + ReifierPatternSet
PatternSet          = { UnaryPattern } + { OperatorPattern } + { KeywordPattern }
ReifierPatternSet   = { ReifierUnaryPattern } + { ReifierOperatorPattern }
                      + { ReifierKeywordPattern }
IdentifierSet       = { Identifier }
OperatorSet         = { Operator }
KeywordSet          = { Keyword }
DelimiterSet        = { Delimiter }

—— productions ——
Message         -> ExpressionSet x PatternSet
ReifierMessage  -> ExpressionSet x ReifierPatternSet

ReifierKeywordPattern       -> (KeywordSet x ExpressionSet)+
ReifierOperatorPattern      -> OperatorSet x ExpressionSet
ReifierUnaryPattern         -> IdentifierSet

KeywordPattern      -> (KeywordSet x ExpressionSet)+
OperatorPattern     -> OperatorSet x ExpressionSet
UnaryPattern        -> IdentifierSet

Aggregate   -> DelimiterSet x ExpressionSet*
```

The class hierarchy that implements Agora's abstract grammar can be found below.

Agora Expression Class Hierarchy

```
AbstractExpression
  MessageExpression (receiver:AbstractExpression, pattern:AbstractPattern)
  ReifierMessageExpression (receiver:AbstractExpression,
                                    pattern:AbstractReifierPattern)
  AggregateExpression (expressions:Array(AbstractExpression),
                              delimiter:Delimiter)
  LiteralExpression (value:LiteralValue)
  PatternExpression (pattern:AbstractPattern)
  ReifierPatternExpression (pattern:AbstractReifierPattern)

AbstractPattern
  UnaryPattern (id:Identifier)
  OperatorPattern (op:Operator, argument:AbstractExpression)
  KeywordPattern (keys:Array(Keyword), arguments:Array(AbstractExpression))
AbstractReifierPattern
  UnaryReifierPattern (id:Identifier)
  OperatorReifierPattern (op:Operator, argument:AbstractExpression)
  KeywordReifierPattern (keys:Array(Keyword),
                              arguments:Array(AbstractExpression))

Identifier (name:String)
Operator (name:String)
Keyword (name:String)
Delimiter (left:String, right:String)
```

Three forms of generic expressions exist: reifier messages, reifier patterns and aggregate expressions; instances of the respective expression classes `ReifierMessageExpression`, `ReifierPatternExpression`, `AggregateExpression`. Their evaluation functions are responsible for dispatching to an appropriate evaluation method. We will not go into the technical details of this mapping. An evaluation method that is invoked due to a reifier message is called a reifier method. We will not go into the details of how reifier methods are represented. We will use a pseudo code instead for introducing new reifier methods (and we will see in the section on reflection that this pseudo code is not to far away from reality).

A *reifier method* is declared as a special sort of method in an expression or pattern class. Since it defines an evaluation method, a reifier method has, in addition to its apparent "subexpression" arguments, a context argument (hidden at the call site of the reifier). This is reflected in the pseudo code by the using clause. An example will illustrate this. The example is the declaration of an assignment reifier method. A reifier method is invoked by an instance of `ReifierMessageExpression`.

```
class UnaryPattern extends AbstractPattern
  methods
    reifier <- (rightHand:AbstractExpression)
        using (context:StandardContext)
      … evaluate the rightHand expression, and store the value
      … in the context, using the receiver as key
endclass

usage in Agora:
a <- 3
```

In a similar vein it is possible to declare reifier classes. A *reifier class* is created and its evaluation method is invoked, every time an instance of the class `ReifierPatternExpression` is evaluated. The dispatcher function for reifier patterns creates an instance of the correct reifier class and then sends an evaluation message to this instance. A reifier class is declared as a special sort of

class. It has an associated pattern (i.e. the name of the reifier pattern that creates it). The associated pattern automatically contains the declaration of the instance variables for subexpressions.

```
reifierclass SelfExpression
    pattern self
    extends AbstractExpression
  methods
    concrete eval:(context:StandardContext)
      … return the current receiver from the context
endclass

usage in Agora:
3 + self
```

Generic aggregate expressions (i.e. generic compound expressions with a variable number of subexpressions) are a straightforward variation of reifier classes. The pattern of an aggregate reifier class contains the delimiters for the aggregate, and a declaration of an "instance variable" for the sequence of subexpressions.

```
reifierclass BlockExpression
    pattern [ exps:Sequence(AbstractExpression) ]
    extends AbstractExpression
  methods
    concrete eval:(context:StandardContext)
      … evaluate each of the subexpressions in the context
endclass

usage in Agora:
[ … ; … ; … ]
```

4.5.2 Message Passing

Like in the implementation of the calculus, the implementation of message expressions plays an eminent role. Unlike the calculus, in the implementation of message passing in Agora, parameter passing must be dealt with. Still, the implementation of message expressions can be done in a way that is independent of evaluation categories.

Agora Message Passing

```
class MessageExpression extends AbstractExpression
  instance variables
   receiver:AbstractExpression,
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandarContext+)
      result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(receiver eval:(context asFunctionalContext))
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

In the description of message passing three things should be noted. One is that arguments are stored in the client object. As announced earlier client objects are used to carry information from the sender object to the receiver. Notice that Agora's client objects are totally unrelated to Simple's client objects due to the lack of an explicit encapsulation operator in Agora.

Agora Standard Client

```
class StandardClient
  public instance variables
    arguments:ArgumentList
endclass
```

Secondly, and more importantly, note the need for casting the context in the course of evaluating expressions. In the above case, contexts are cast to ensure that message expressions can be evaluated in all possible evaluation categories. When evaluating a message expression, the context in which the entire message expression is evaluated, and the context in which the receiver and argument expressions are evaluated can not be (exactly) the same since the receiver and arguments always have to be evaluated in a functional evaluation category. On the other hand, the context in which the receiver and arguments of a message expression are evaluated must be derived from the context in which the entire message expression is evaluated. So the context in which message expressions are evaluated is cast to a functional context for the evaluation of receiver and arguments. The protocol of context objects is adapted accordingly.

Agora Standard Context (Extract)

```
class StandardContext
  public instance variables
    ...
  methods
    abstract asFunctionalContext result StandardContext
      ... return an instance of functional context with the same
      ... content
endclass
```

The need for casting contexts is not limited to message expressions. Due to Agora's extremely simple syntax, evaluation categories play an eminent role, and thereby also the need for expressing dependencies between evaluation categories.

Finally, a word is in order about the role of patterns in the implementation of Agora. Unlike patterns in the calculus, Agora patterns that are part of some program representation, include argument expressions. For this reason an uncoupling of message patterns that are part of a program representation, and patterns that are used at run-time is in order. The latter sort of patterns (instances of the class `AbstractCategoryPattern`) are mainly used as unique identifiers. Patterns that are part of a program representation are turned into patterns that can be used for message passing via the `asCategory` message, i.e. this transformation process must take evaluation categories into account.

Patterns Used in Expressions Versus Patterns Used in Messages

```
class AbstractPattern
  methods
   abstract asCategory:StandardContext+
     result AbstractCategoryPattern
endclass
```

```
class AbstractCategoryPattern
  methods
   abstract = AbstractCategoryPattern+ result Boolean
endclass
```

Apart from message expressions with an explicit receiver, Agora also has message expressions with an implicit receiver. These are the so called *receiverless message expressions*. They are represented as instances of the `PatternExpression` class. The implementation of this class is similar to the implementation of message expressions, except for the fact that the receiver is a predefined part of the context.

Evaluation of Receiverless Messages

```
class PatternExpression extends AbstractExpression
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandaredContext+)
      result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(context privatePart)
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

4.5.3 Mixin Application and Object Structures

Agora objects differ from the previously discussed objects, from the calculus, in different ways. Although the essential protocol of message passing is the same, Agora objects have a more complex internal structure. Furthermore, since Agora is essentially a language with side-effects, issues such as object equality and object cloning must be dealt with. Calculus objects are richer in one way, the encapsulation operator on objects, that plays such an eminent role in the calculus, is not present in Agora. We will see that internally, Agora objects will use an operator reminiscent of Simple's encapsulation operator. The difference is that the latter is an encapsulation operator on objects, and the former is more comparable to an encapsulation operator on generator functions as discussed in the previous chapter.

A note should be made about the relation of the framework with (nested) mixin method inheritance. Mixin method inheritance is important for Agora, and it is this form of inheritance that will be discussed in the framework. The question then arises whether it is possible to implement other inheritance mechanisms in the framework. This question can not be answered with a convincing 'yes'. Clearly the fact that interaction with objects is limited to message passing (and not delegation for example) is a serious constraint in this context. Since with mixin methods, inheritance is based upon sending messages to objects this is no

problem. For other inheritance mechanisms such a strong encapsulation probably is a problem. Accordingly implementing such a mechanism in this framework will involve extending the framework outside its intended usage, i.e. it will involve extending the framework in a less reusable fashion.

4.5.4 Agora Internal Object Structure

As we already said before, there exists a plethora of different kinds of objects. Still, we have opted for, and briefly discussed the advantages of, an abstract object representation. In casu, objects that can be sent messages. All implementation details of objects remain hidden in the object representation. This must be equally true for the inheritance structure of objects. It should not become apparent in an objects representation whether it inherits (or should inherit) from another object.

Unlike Simple objects, Agora objects have a complex internal structure. On the one hand this structure must be hidden in the object representation, on the other hand, with the eye on extensibility, a complex, and ad hoc, object structure needs to be avoided.

A solution was found in encoding the object structure as a structure of finer grained internal objects that communicate with each other with (variants of) delegation. This solution is based on the fact that it is possible to mimic almost any inheritance structure with a delegation based system [Lieberman86]. As we will show in a moment, the notion of delegation is extended to take mixins and objects with structured private attributes into account.

The general idea is to delegate messages to objects in an explicitly given context. This context not only encodes how message passing must proceed, but also how the body of a slot must be interpreted once it has been found. For example, the context can have a field that stores the original receiver of the object for interpreting future "self expressions" (which is the original meaning of delegation); it can have a "parent" field that will either be used to further delegate a message if no slot is found in the current object that corresponds to the message, or it is used for interpreting "super expressions" in the evaluation of the slot if it is found; or it can have an "encapsulated part" field that is used, amongst other things, as the receiver of all receiverless messages (in casu identifier lookup) that occur in the evaluation of the body of the found slot.

This will give rise to a set of different (internal!) objects that can be flexibly combined through delegation. The subset used in the implementation of Agora is given below. It will be shown how this set of internal objects is used to implement mixin based inheritance. Abstractly all internal objects respond to a delegate message that has an extra delegation context argument. This latter argument encodes all extra delegation information.

Root of the Abstract Internal Object Classes

```
class AbstractInternalObject
  methods
    abstract  delegate:CategoryPattern
              client:StandardClient
              context:DelegationContext+
        result AbstractMetaObject
endclass
```

A delegation context is a context in which extra fields can be filled in. The two extra fields used in the delegation structure for the encoding of mixin methods in

Agora are a parent field and a private field.

While delegating a message the parent field contains the parent object of the object that receives the delegation request. In fact when an object that is not composed of subobjects receives a delegated message that it does not want to reply to, it can use the parent field to further delegate the message.

The private field is used in the encoding of objects that have encapsulated attributes. Again, an object that is not composed of subobjects and that accepts a delegated message can use the private part of the delegation context to evaluate its method bodies in.

Contexts Used in the Realisation of Mixin methods

```
class DelegationContext extends StandardContext
  methods
    …
    concrete parent:AbstractInternalObject result DelegationContext
      —— returns a copy of the context with a new parent field
    concrete private:AbstractInternalObject result DelegationContext
      —— returns a copy of the context with a new private field
    concrete noParent result DelegationContext
      —— returns a copy of the context with a empty parent field
    concrete noPrivate result DelegationContext
      —— returns a copy of the context with a empty private field
    …
endclass
```

```
class ParentContext extends DelegationContext
  public instance variables
    parent: AbstractInternalObject
endclass
```

```
class EncapsulatingContext extends DelegationContext
  public instance variables
    private: AbstractInternalObject
endclass
```

Agora's inheritance structures are constructed by means of instances of the following three internal object classes. The first class encodes objects that have a parent object. On reception of a delegated message, this message is forwarded to the object that contains the locally defined attributes (i.e. `thisPart`). It is forwarded such that the parent object (the `parentPart`) is recorded in the delegation context. The local part of the object will forward the message to the parent if it does not respond, itself, to the message.

Concrete Internal Object Classes: 1) Objects with a Parent

```
class ObjectWithParent extends AbstractInternalObject
  instance variables
    thisPart: AbstractInternalObject
    parentPart: AbstractInternalObject
  methods
    concrete delegate:pattern client:client context:(context:DelegationContext+)
        result AbstractMetaObject
      ^thisPart delegate:pattern
                client:client
                context:(context parent:parentPart)
endclass
```

The second class encodes objects that have encapsulated attributes. Similar to above a message is forwarded to the object that contains the public attributes

(the publicPart) in a delegation context that records the private part of the object. The selected attribute in the public part will be evaluated in a context that is built up with the private attributes found in the delegation context.

This class resembles the CompoundObject class from the implementation of Simple. The difference is that here attributes are not encapsulated into an object after the object was created by an explicit encapsulation operator, but rather they are declared encapsulated when the object is created. This difference has already been discussed in the section on encapsulation operators for objects in the previous chapter.

Concrete Internal Object Classes: 2) Objects with Encapsulated Attributes

```
class EncapsulatedObject extends AbstractInternalObject
  instance variables
   publicPart: AbstractInternalObject
   privatePart: AbstractInternalObject
  methods
   concrete delegate:pattern client:client context:(context:DelegationContext+)
       result AbstractMetaObject
     ^publicPart
         delegate:pattern
         client:client
         context:(context private:privatePart)
endclass
```

The last class encodes objects that have no further subobjects, but directly store slots. These objects are important in this discussion since they will use the information in the delegation context to respond to delegated messages. First of all they will use the parent field of the delegation context to further delegate a message when this message is not locally handled. Secondly, they combine all other information found in the client (i.e. the information coming from the sender), and the information from the delegation context. This combined information is used to evaluate selected slots in. The parent field of the delegation context, for example, will be used to interpret parent operation invocations.

Concrete Internal Object Classes: 3) Objects without Subobjects

```
class SimpleObject extends AbstractInternalObject
  instance variables
   thisPart:Sequence(Slot)
  methods
   concrete delegate:pattern client:client context:(context:ParentContext+)
       result AbstractMetaObject
     slot := thisPart findSlot:pattern
     if slot
        then ^slot valueIn:(context with:client)
        else ^(context parent) delegate:pattern
                               client:client
                               context:(context noParent)
endclass
```

The above internal objects can be combined in different ways. The corresponding delegation contexts must be combined likewise. Delegation contexts can be combined with an appropriate multiple inheritance mechanism. The exact details are not important. What is important is that the delegation context in an actual implementation will be a combination of the two above listed and all other delegation contexts defined further on in the text.

Agora's inheritance structure can be used to illustrate how the above internal objects can be combined in a useful way. In the construction of nested mixin methods, objects with parents are not only used to link all public parts of an object, but also to link all encapsulated parts of an object. This can best be illustrated with an example. Figure 4.11 shows the internal representation of the objects of the following Agora program. This figure is best interpreted from right to left. The wrapper objects, totally on the right will be explained in the next section.

```
[ MakeX Mixin:
    [ xiv define ;
      xm Method: [ xiv <- 3 ] ;

      MakeY Mixin:
      [ yiv define ;
        ym Method: [ xiv <- 4 ; yiv <- 8 ] ] ;

      MakeZ Mixin:
      [ ziv define ;
        zm Method: [ xiv <- 5 ; ziv <- 9 ] ] ] ;

  Root define: self ;
  X define: Root MakeX ;
  XY define: X MakeY ;
  XYZ define: Y MakeZ ]
```

Each mixin-application adds a set of public attributes and a set of private attributes to the receiving object. Correspondingly each mixin application results in the creation of a new 'layer' in the hierarchy of internal objects. All layers are linked by instances of `ObjectWithParent` (the linked chain of objects on the right in the figure). Within each layer the private attributes are associated to the public attributes with an encapsulated object. Finally, all encapsulated objects form a hierarchy built up, again, with objects of class `ObjectWithParent`. This hierarchy encodes the nesting structure of the mixins in the above program.

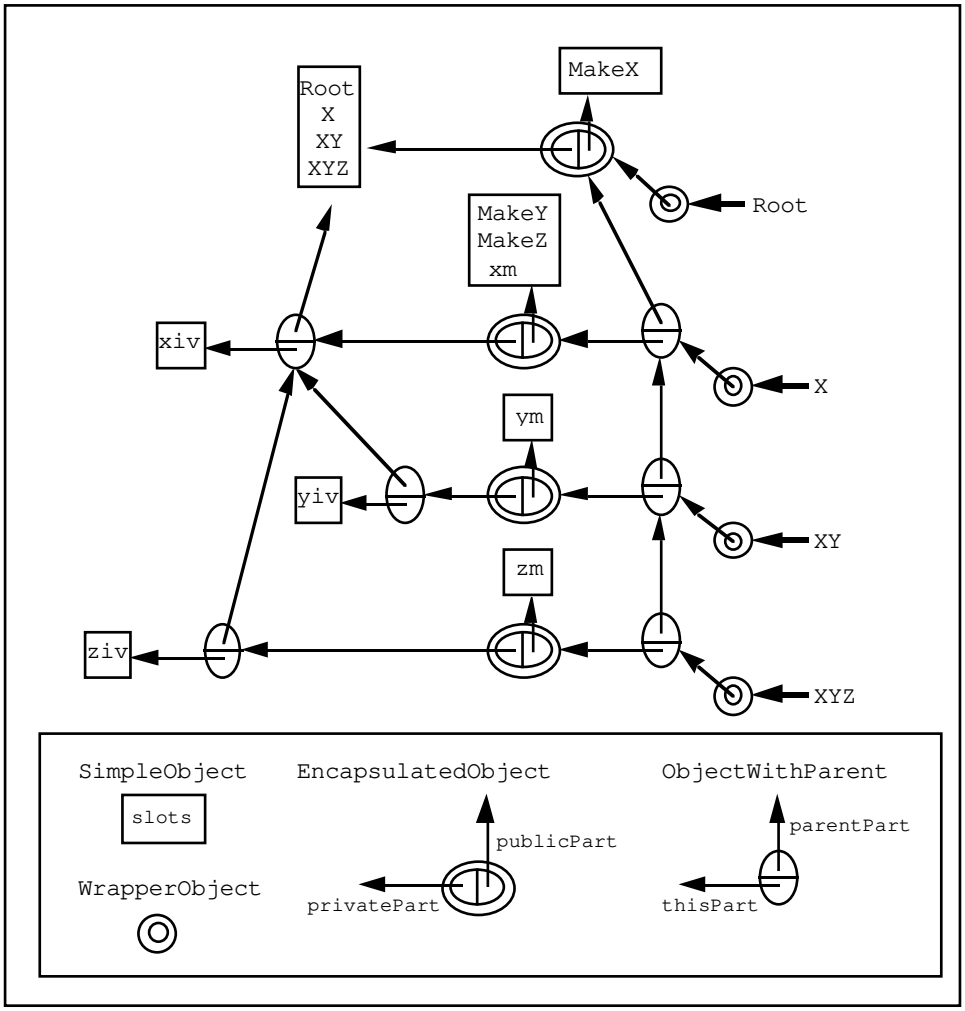

Figure 4.11

The way the scoping of nested mixins is dealt with is worth noting. The scoping of nested mixins is totally resolved with inheritance of encapsulated objects. In the figure both Z, and Y inherit in their encapsulated part from X, without inheriting from each other.

4.5.5 External Object Structures and Wrapper Objects

Internal objects can not be used directly in the evaluation of expressions. They do not hide enough details of their implementation, as opposed to meta-objects that only respond to the pure message passing protocol. Internal details of Agora internal objects are hidden by *wrapper objects*. Wrapper objects serve two purposes: 1) they act as holders of internal objects, thereby hiding their internal details 2) they are the explicit identity of objects. Wrapper objects are the only kind of meta-objects in use in the implementation of Agora. All variations on objects in Agora are due to variations in the internal object structures.

A wrapper object is essentially a forwarder of messages to its wrapped object. All accepted messages are delegated to the wrapped object, that is stored in the `delegate` instance variable. The wrapped object can be any constellation of internal object structures. Wrapper objects are responsible for generating recursive object structures, i.e. wrapper objects record themselves in the delegation context as receiver object. The `receiver` field of the delegation context can be used to interpret self expressions. The wrapped object is also put in the delegation

context. This has to do with the mechanism to extend a receiver object with mixins, as will be shown in the next section. A specialised delegation context with a `receiver` field and a `public` field is used in the implementation of wrapper objects.

Wrapper Objects

```
class WrapperObject extends AbstractMetaObject
  instance variables
   delegate: AbstractInternalObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^delegate delegate:pattern
               client:client
               context:(DelegationContext receiver:self
                                          public:delegate)
endclass
```

Agora Delegation Contexts that Record the Receiver

```
class DelegatingContext extends DelegationContext
  public instance variables
   receiver: AbstractMetaObject
   public: AbstractInternalObject
endclass
```

Wrapper objects encode the identity of Agora objects. As stated before, since Agora is an imperative programming language testing objects for identity is an important operation. Internally objects can have shared structures, but each object in Agora is represented by a unique wrapper. This allows dramatic changes to the internal object structures without changes to the identity of an object. An identity swap operation could easily be defined, for example. It suffices for two wrapper objects to swap their (private) wrapped objects. Other, more constructive, examples such as object reclassification can be implemented as easily. Note that it is this kind of variations that are the useful variations on wrapper objects.

For simplicity reasons we also expect, in the remainder of the text, that all internal objects have a wrap method. This wrap method puts a wrapper object around the receiving internal object, or in general hides the internal details of that object.

4.5.6 Extending Objects, Execution of Mixin Methods

Agora objects are extended by executing mixin methods. A mixin method is like an ordinary method except that its body is evaluated in a special mixin evaluation category (this is by the way, a good example of the usage of evaluation categories). Only block-expressions evaluate in the mixin evaluation category. They do so by extending the receiver with a new public and private part, and evaluating all the component expressions in this new receiver.

The evaluation of blocks in the mixin evaluation category is listed below. The mixin context used in this implementation indicates this evaluation category. This description may seem a bit involved, but what it actually does is adding an extra layer to the internal object structures to store public and private slots.

mixin method execution

```
reifierclass BlockExpression
   pattern [ exps:Sequence(AbstractExpression) ]
   extends AbstractExpression

  abstract class attributes
   ExtensibleSimpleObject ObjectWithParent EncapsulatedObject
  methods
   concrete eval:(context:MixinContext) result AbstractMetaObject
     local variables privateSlots publicSlots privatePart
              myPart publicPart newReceiver newContext
     privateSlots := ExtensibleSimpleObject new
     publicSlots := ExtensibleSimpleObject new
     privatePart := ObjectWithParent thisPart:privateSlots
                                     parentPart:(context private)
     myPart := EncapsulatedObject publicPart:publicSlots
                                  privatePart: privatePart
     publicPart := ObjectWithParent thisPart:myPart
                                    parentPart:(context public)
     newReceiver := publicPart wrap
     newContext := (context asImperativeContext)
                       privateSlots:privateSlots
                       publicSlots:publicSlots
                       receiver:newReceiver
     for each exp in exps do
       exp eval:newContext
     ^newReceiver
endclass
```

Unlike Simple, Agora objects are not created by first collecting their slots and then creating an object with this collection of slots. In Agora slots are added to an object by means of declaration reifiers (e.g. method declaration, variable declaration, etc.). Declarations and other expressions may be mixed. For this reason, it must be possible to add new slots to the receiver after it has been created. This is possible with the following extension to the `SimpleObject` class of internal objects. With this extension slots can be added to this particular kind of internal objects. Also the mixin evaluation context must have two fields, each containing a reference: one to the object in which the public slots of the current extension are stored, and one to the object in which the private slots are stored.

Adding Slots to Agora Internal Objects

```
class ExtensibleSimpleObject extends SimpleObject
  methods
   concrete add:slot
     slots add:slot
endclass
```

Agora Delegation Contexts that Record the Public and Private Slots

```
class MixinContext extends DelegationContext
  public instance variables
   publicSlots: ExtensibleSimpleObject
   privateSlots: ExtensibleSimpleObject
endclass
```

Notice however that only the public and private slots of the *current* receiver object can be extended. The context only contains references to the extensible objects of the current receiver. All other objects can only be accessed via their wrapper, and they can only be extended by sending mixin messages (i.e. messages that result in the execution of a mixin method). Encapsulation is preserved !

4.5.7 Object Cloning

Object cloning plays an important role in Agora (as in any object-based programming language). In some form or another one can expect a cloning operation for objects. This might take the form of a simple clone method on objects, or, as we will see later on, more sophisticated constructions are possible.

Given the nature of how objects are represented internally, it is not evident how cloning must be implemented. First of all, internal objects use a non hierarchical sharing structure (e.g. in the realisation of nested scoping), this sharing structure must be preserved after cloning. Moreover, the cloning mechanism must be extensible, i.e. it must be possible to add new internal objects with their associated cloning strategy.

The cloning strategy for internal objects is made flexible through the introduction of *clone maps*. Clone maps (or a variant thereof) are typically used when copying circular pointer structures. They record the objects already copied, and associate each original object with its copy. It is obvious that this information can be used to copy objects while preserving circular or shared references.

Clone maps provide the classical operations for object cloning. A clone map can be asked to deep copy an object. Objects already present in the map will not be copied, rather the associated object in the map will take the place of the copy. A map can be asked to shallow copy an object. If this object is already in the map, the associated object is returned.

Clone Maps

```
class CloneMap
  methods
    concrete shallowClone:AbstractClonableInternalObject
        result AbstractClonableInternalObject
      … take a shallow clone of the argument if not present
      … in the map
    concrete deepClone:AbstractClonableInternalObject
        result AbstractClonableInternalObject
      … take a deep clone of the argument preserving the sharing
      … structure
      … all objects in the map will not be cloned
      … the map is extended with the newly copied objects
    concrete shallowNoClone:AbstractClonableInternalObject
      … extend the map with an identity association on the argument
    concrete deepNoClone:AbstractClonableInternalObject
      … extend the map with identity associations of all the direct
      … and indirect acquaintances of the argument
endclass
```

In our case clone maps are used to achieve a general and flexible object cloning mechanism. All clonable internal objects must provide three sorts of cloning operations. An object's outline (i.e. acquaintances are not copied) is copied with the shallow copy operation. Deep cloning is used to copy an object's acquaintances while respecting the clone map.

Abstract Class for Clonable Internal Object Classes

```
class AbstractClonableInternalObject extends AbstractInternalObject
  methods
    abstract shallowClone result AbstractClonableInternalObject
    abstract deepClone:CloneMap result AbstractClonableInternalObject
    abstract deepNoClone:CloneMap
endclass
```

An example of how compound objects are cloned will illustrate the above.

```
class ClonableObjectWithParent extends
                    AbstractClonableInternalObject,
                    CompoundObject
  methods
    concrete shallowClone result AbstractClonableInternalObject
      ^ClonableObjectWithParent
          publicPart:publicPart
          privatePart:privatePart.
    concrete deepClone:cloneMap result AbstractClonableInternalObject
      publicPart := cloneMap deepClone:publicPart
      privatePart := cloneMap deepClone:privatePart
    concrete deepNoClone:CloneMap
      cloneMap deepNoClone:publicPart
      cloneMap deepNoClone:privatePart
endclass
```

Finally note that, for now, clone maps are only used when copying the internal structure of objects. It is often desirable to use clone maps on the level of objects themselves, or even provide clone maps at the language level. Although this seems no problem in principle, this was not our initial motivation for the introduction of clone maps, and we did not further investigate this possibility. We refer the reader to [Mittal,Bobrow,Kahn86] for this matter.

4.5.8 Mixin, Method and Instance Variable Declaration Reifiers and Slots

A set of reifiers has been defined for adding slots to objects. Listed below are the mixin and other method declaration reifiers. Their implementation is straightforward. They just add a slot to the public part of the receiver object. This slot associates the method pattern to the method body.

Method and Mixin Declaration Reifiers on Patterns

```
class AbstractPattern
  abstract class attributes
    MethodSlot MixinSlot
  methods
    reifier Method:(rightHand:AbstractExpression)
        using (context:MixinContext)
      context publicSlots add:
        (MethodSlot key:self value:rightHand)

    reifier Mixin:(rightHand:AbstractExpression)
        using (context:MixinContext)
      context publicSlots add:
        (MixinSlot key:self value:rightHand)
endclass
```

Reifiers for declaring variables — either local variables or instance variables — are restricted to unary patterns. Only the define reifier has been listed below, all other variations have a similar implementation. A variable declaration adds two accessor slots to the private part of the receiver: one for reading the

variable, one for writing the variable. These slots share a reference to a variable holder that stores the value of the variable. As mentioned in the introduction to Agora, the assignment reifier is interpreted as a message that is sent to the private part of the receiver object. The equivalent message of assigning for example the value `3` to an identifier `x` is the receiverless message `x:3`.

Variable Declaration and Assignment Reifiers on Unary Patterns (Identifiers)

```
class UnaryPattern extends AbstractPattern
  abstract class attributes
   VariableHolder ReadVariableSlot WriteVariableSlot
  methods
   reifier define using (context:ImperativeContext)
     variableHolder := VariableHolder new.
     context privateSlots add:
       (ReadVariableSlot
         key:(self asFunctionalCategoryPattern)
         value:variableHolder)
     context privateSlots add:
       (WriteVariableSlot
          key:(self asImperativeCategoryPattern)
          value:variableHolder)

   reifier <- (rightHand:AbstractExpression)
       using (context:ImperativeContext)
     context privatePart
       send:(self asImperativeCategoryPattern)
       client:(rightHand eval:(context asFunctionalContext))
endclass
```

The slots that are used in Agora, have the same functionality as slots in the calculus. We will not go into the details of all the different slots introduced in the above description. Their implementation is a straightforward extension of previously defined slots. The context in which slot-bodies are evaluated in Agora are a direct derivation of the context used in delegating messages. This is logical. The body of a method is evaluated in a context that is essentially the receiver object.

4.5.9 Summary of the Application of the Framework to Agora

Whereas the implementation of Simple was used to improve our initial proposal for a framework (e.g. the introduction of client and context objects, the introduction of evaluation categories), the implementation of Agora indicates refinements and extensions to this improved framework.

The most important refinement is the introduction of internal object structures. This adds an extra layer to the framework. It is a partial concretisation of how meta-objects can be implemented. The notion of wrapper objects can be important for the implementation of flexible imperative objects. The framework was extended with notions such as object cloning. It was shown that a cloning facility can be constructed while preserving flexibility in the internal representation of objects.

Most importantly it was shown that the framework, albeit simple in nature, is general enough to form the basis for the construction of a full-fledged object-oriented language. It is also important to note that the notions of reifier methods and classes were consistently used for the entire definition of Agora, except for message passing. Message passing, which forms the kernel of Agora is the only built-in language construction.

4.6 Extensions to Agora

The framework introduced in the previous section can be used to define a set of extensions to Agora. The purpose of this section is to illustrate the flexibility of the framework. In our previous discussions we encountered an entire range of language concepts that should be supported, either for the construction of frameworks or for the construction of multiple inheritance, …. A selection of such language concepts is presented below. They are expressed in the framework.

4.6.1 Public Instance Variables and Private Methods

The standard set of reifiers for Agora does not include the declaration of neither *private methods* nor *public instance variables*. This is a straightforward extension however, due to two facts. One is that instance variables are represented as a "get instance variable", and a "set instance variable" slot, and that reading and writing instance variables is done through message passing. The second fact is that we already provide a mechanism to provoke private methods, i.e. receiverless messages. Their implementation can be found in the previous section.

```
[ MakeX Mixin:
    [ xiv publicdefine ;
      testPrint:aString PrivateMethod: [ aString print ] ;
      test Method: [ testPrint:"test" ] ] ;

  X define: Object MakeX ;
  X xiv:4 ;
  X xiv print                                  —— 4 on transcript
  X test                                       —— "test" on transcript
]
```

The implementation of the declaration reifiers for public instance variables and private methods is straightforward.

Agora Extension: Private Method Declaration Reifier

```
class ExtendedAbstractPattern extends AbstractPattern
  methods
    reifier PrivateMethod:(rightHand:AbstractExpression)
        using (context:StandardContext)
      context privateSlots add:(MethodSlot key:self value:rightHand)
endclass
```

Agora Extension: Public Instance Variable Declaration Reifier

```
class ExtendedUnaryPattern extends UnaryPattern
  methods
    reifier publicdefine using (context:StandardContext)
      variableHolder := VariableHolder new.
      context publicSlots add: (ReadVariableSlot key:self value:variableHolder)
      context publicSlots add:
        (WriteVariableSlot key:(self asImperativeCategoryPattern)
                           value:variableHolder)
endclass
```

4.6.2 Cloning Methods

In object-based programming languages there is a need for sophisticated cloning operations (see for example [Mittal,Bobrow&Kahn86]). Up until now we only discussed a simple clone reifier in Agora. As an example of a more sophisticated cloning operation we will show an extension of Agora in which cloning and initialisation of objects are combined.

For encapsulation reasons it is often desirable to combine initialisation of the private state of an object and copying of that object. In pure object-based languages, where new instances can only be made by copying old instances, one usually needs to initialise the newly created instance after copying it. This is done with an initialisation method. In most cases this initialisation method must only be invoked on a newly created instance, but in most languages this is not enforced. The same problem occurs in class-based languages (in Smalltalk for example, as a convention between Smalltalk programmers, a special message category of "initialisation methods" or "private methods" is reserved for this purpose, in C++ a special copy constructor mechanism is available).

We propose the following alternative where a special category of methods, the category of *cloning methods*, is reserved for cloning objects. A cloning method contains initialisation code. When a cloning method is invoked, it will be invoked on a copy of the receiver object.

```
[ MakeX Mixin:
    [ xiv define ;
      xiv:newx CloningMethod: [ xiv <- newx ] ;
      xiv Method: xiv Result:Integer ] ] ;

  X define: Object MakeX ;

  y define: X xiv:3 ;
  z define: X xiv:5 ;
  y xiv print ;                    —— 3 on transcript
  z xiv print ;                    —— 5 on transcript
]
```

The implementation of cloning method slots is less straightforward than can be expected. This has to do essentially with the scoping rules of Agora and the way messages are delegated (rather than being looked up) internally. On reception of a message an object does not know whether this will result in the execution of an ordinary method or for example a cloning method. So, it can not decide at this point whether to proceed with a copy of itself or not. This decision can only be made when a method is found. The point is that by then the (delegation) context contains parts of the receiver object that is to be copied. Consequently the receiver object can not be copied as a whole, but needs to be copied part by part, whereby all parts need to be assembled. Cloning maps are a very suitable solution for this problem.

Agora Extension: Cloning Methods

```
class ExtendedAbstractPattern extends AbstractPattern
  methods
   reifier CloningMethod:(rightHand:AbstractExpression)
       using (context:StandardContext)
     context privateSlots add:
       CloningMethodSlot key:self value:rightHand
endclass
```

```
class CloningMethodSlot extends MethodSlot
  methods
   concrete valueIn:(context:StandardContext)
     local variables aMap parentPart privatePart publicPart
                     receiver
     aMap := CloneMap new.
     parentPart := (aMap clone:context parentPart).
     privatePart := (aMap clone:context privatePart).
     publicPart := (aMap clone:context publicPart).
     receiver := (publicPart wrap).
     super valueIn: (context  parentPart:parentPart
                              privatePart:privatePart
                              publicPart:publicPart
                              receiver:receiver).
     ^receiver
endclass
```

The positive point about this implementation is that it has some interesting variations. In some cases it is not desirable that the entire receiver is copied. One variation is that the receiver is only copied up to the point where the cloning method is found. This can easily be realised. Instead of copying the parent part from the context (i.e. `parentPart := (aMap clone:context parentPart)`), the parent part is inserted 'as is' in the cloning map (i.e. `aMap noClone:context parentPart`). Such variations are useful for the introduction of shared instance variables or when handling 'split objects'.

4.6.3 Stubs for Multiple Inheritance

In the section on mixins with multiple parents, a *message qualification mechanism* for mixins was discussed. It was based on the notion of inserting *stubs* in the inheritance chain. This mechanism can be adapted for Agora. The car-toy example that illustrated the usage is translated to Agora, extended with stubs, as follows:

```
[  makeCar  Mixin:
  [ fuel publicdefine:"gasoline" ;
    print Method:[ self fuel print ] ] ;

   makeToy Mixin:
    [ age publicdefine:2 ;
      print Method:[ self age print ] ] ;

   CarStub Stub:
    [ ToyStub Stub:
      [ MakeCarToy Mixin:
        [ print Method: [  CarStub super:print ;
                           ToyStub super:print ] ] ]
] ;

Car <> Object makeCar ;
Toy <> Object makeToy ;
ToyCar <> Car CarStub makeToy ToyStub MakeCarToy ;
aToyCar <> ToyCar ;
aToyCar print  ]
```

Notice that to stay in Agora's philosophy, stubs are declared in the same way that methods and mixins are declared (they can even be nested as in the above example). A stub is inserted in between two mixin applications by sending a corresponding (stub) message. An object, upon reception of a message that leads to the selection of a stub-slot, inserts that stub in its inheritance chain. A special super invocation reifier is provided that takes stubs into account, i.e. to invoke operations of non-direct parents.

Remark that the above example can also be interpreted in a less operational manner. A mixin declares the number and formal names of its possible parents (with the stub declarations). These formal names can be used in parent invocations. The formal names are bound to actual parents in the mixin application chain by inserting references to the formal parent names (with stub applications).

A second remark is about the role of nesting stubs and mixins. A stub name serves two purposes. It is used in the declaration of a public stub attribute with which the stub can be inserted in the inheritance chain. When inserted it serves as the name of a private attribute used in the parent invocation. Since private names are lexically scoped in Agora, mixins must be nested in the stub declarations of the stubs they want to use in their parent invocations. In the above example, the nesting of stubs imposes an order on the mixin-applications with which car-toys can be made. First a car must be made and then, and only then, this car can be extended to a toy-car. An alternative stub declaration in which the order of stub-application is free, can be devised. It would take the form:

```
{CarStub, ToyStub} Stub:
   [ MakeCarToy Mixin: [ … ] ]
```

The notion of stubs fits well in the framework. Still, extending the framework with stubs is a bit more complicated than the previous extensions. We will confine ourselves to a brief overview.

Agora must be extended with two new reifiers: a reifier to declare stubs and a reifier to invoke operations of non-direct parents. Upon evaluation, the stub declaration reifier inserts a 'stub-slot' in the public part of an object. This stub slot can be invoked by a (stub) message. The effect is that the receiving object is extended with a stub-object. This stub-object is set to contain, in a private slot, a reference to the object to which it is applied. The contents of this private slot is used by the parent invocation reifier to delegate messages to (a parent operation invocation is implemented by delegation !). Therefore the stub-object must contain a reference to the internal representation of the object that it refers to.

The above implementation involves a number of technicalities. How can the stub-object get a hold on a non-encapsulated version of the receiver object (in fact the internal representation of the receiver) ? How can we avoid that stub-objects are passed around and see to it that they are only used in parent invocations ? How do stub-objects influence object cloning ? We will not go into the details. We only note that it is possible to solve them within the constraints of the framework.

4.6.4 Single Slot Nested Objects

The standard flavour of Agora includes a fixed set of control structures. In object-oriented programming languages the construction of user-defined control structures is an important issue. In general a derivative of closures is employed for this purpose. The extension of Agora with *single slot nested objects* goes along these ways. The idea is to create objects with a single slot (objects that respond to only

one message), and that are dependent on their creation context. An example is given below. It is the classical object-oriented definition of boolean values. The '@' reifier combines a pattern and a body for that pattern to a single slot nested object. The example only features single slot objects that respond to unary messages, in general operator and keyword patterns can also be used in the creation of single slot objects. Standard argument passing applies for single slot objects.

```
[  makeTrue  Mixin:
   [ ifTrue:tBlock ifFalse:fBlock Method:[ tBlock true ]
   ] ;

    makeFalse  Mixin:
   [ ifTrue:tBlock ifFalse:fBlock Method:[ fBlock false ]
   ] ;

   True <> Object makeTrue ;
   False <> Object makeFalse ;

   aBoolean define ;

   aBoolean <- True ;

   aBoolean  ifTrue:(true@["this" print])
             ifFalse:(false@["that" print])
]
```

Again, we will only give a brief overview of how the framework must be extended. Single slot nested objects are best regarded upon as a variation of wrapper objects. They are objects that reinterpret the notion of self reference and private attributes. Single slot objects inherit their 'self' and their private attributes from the surrounding context. This is the basis for their implementation. They are wrapper objects that contain a reference to the context in which they are created, and a reference to a single slot. Upon reception of the appropriate message, the body of this single slot is evaluated in the stored creation context. Single slot nested objects are a good example of the useful variations on wrapper objects.

4.6.5 Classes

Agora is at its heart a prototype based programming language. Still, following the analysis of a previous section, classes can be reintroduced. It suffices to make a distinction between 'class' objects and 'instance' objects. In the following example this distinction is introduced by the 'class' reifier. Variables declared with the class reifier can only contain class objects. Class object can only be sent mixin and cloning messages. A clone of a class object is an instance object. Instance objects can not be sent mixin nor cloning messages.

```
MakePerson Mixin:
   [ name define ;
     name:newName CloningMethod:[name <- newName] ;
     name Method:name ] ;

MakeSportsman Mixin:
   [ cardnumber define ;
     number:newNr CloningMethod:[cardnumber <- newNr ] ;
     number Method:cardnumber ] ;

Person class: Object MakePerson ;
SportsPerson class: Person MakeSportsman ;

john define: Person name:'John';
```

Class objects are another good variation of wrapper objects. They are wrapper objects that filter the accepted messages. They do so by heavily relying on the notion of pattern-categories. All messages received by a class object are delegated to the internal object structures in a special pattern category that is only compatible with mixin and cloning patterns. All messages received by an instance object are delegated to the internal object structures in a special pattern category that is *not* compatible with mixin and cloning patterns. Accordingly, a mixin message sent to an instance object results in a 'message not understood' error.

4.6.6 Abstract Methods, and Abstract Class Attributes

Obviously the introduction of *abstract methods* is a very easy extension of the framework. It suffices to define an abstract method declaration reifier that stores an abstract method slot in the public part of an object. This slot responds with an error when selected. Or, even better, the cloning of objects could be adapted such that an object with an abstract method slot returns an error when cloned. Concretisation of an abstract method relies on method overriding. In a statically typed variant of Agora the information provided by an abstract method declaration could be useful.

```
MakeButton Mixin:
  [ draw:window AbstractMethod ]
```

The introduction of *abstract class attributes* is less trivial. Concretisation of abstract class attributes relies on overriding of private attributes. Since in standard Agora private attributes are lexically scoped (and not dynamically) this proves to be a problem. In fact, the lack of overridable private attributes is an open problem for Agora. Preliminary investigations have shown that the solution is closely related to introducing an explicit encapsulation operator in Agora (such as can be found in Simple), thereby simplifying the entire structure of internal and context objects. This needs further attention.

4.6.7 A Simple Form of Monotonic Reclassification

Mixins can be applied to objects. The result is a new extended object, that shares the parent object with all other such extensions. In some cases an object must be extended without resulting in a new object. This is a form of *monotonic reclassification*. Consider the following example. A person john becomes a doctor. Note that john truly *becomes* a doctor: all the references to john see john as a doctor after the object that represents john is extended. In an extension of Agora this is realised by applying a mixin in an imperative manner (previously all mixin applications returned a result: the extended object).

```
MakePerson Mixin:
  [ name define ;
    name:newName CloningMethod:[name <- newName] ;
    printname Method:[name print] ] ;

MakeDoctor Mixin:
  [ printname Method:['Dr. ' print ; super:printname] ] ;

Person define: Object MakePerson ;

john define: Person name:'John' ;
johny <- john ;

john MakeDoctor ;

johny printname              -- prints Dr. John
```

Wrapper objects play an important role in the implementation of the above imperative mixins. The idea is to extend the internal object structures of an object while keeping the same wrapper object. For the rest normal mixin application does it.

The above is a simplified form of *reclassification*. It allows for an object to gain new attributes. More powerful mechanisms are imaginable. A similar notion to stubs, for example, could be used as marker points to drop attributes from an object.

4.6.8 Classifiers

Mixins tend to split up the inheritance hierarchy into small chunks of behaviour. Generally, the number of attributes declared in a mixin is much smaller than the number of proper attributes declared in a class with a 'plain' inheritance mechanism. Therefore mixins form highly combinable primitives for the construction of objects. On the other hand a mechanism is needed to manage this combinatorial explosion. In the multiple inheritance literature the notion of *classifiers* [Hamer92] or *inheritance dimensions* [McGregor&Korson93] has already been proposed for this purpose. Classifiers can be easily adapted for mixin methods.

Consider the following example. The fact must be recorded that the mixin to turn a person into a female can not be combined with the mixin to turn a person into a male. The notion of gender is introduced at the level of persons. It is formally declared that a person can be classified according to its gender (i.e. a person can have a gender dimension in her/his inheritance chain), but can only be classified once according to the gender. Both the `MakeFemale` and the `MakeMale` mixins subscribe to the gender classifier. An attempt to apply both to the same person will result in an error.

```
MakePerson Mixin:
  [ gender ExclusiveClassification ;
    …
  ] ;

MakeFemale Mixin:
  [  Classification:gender ;
    …
  ] ;

MakeMale Mixin:
  [  Classification:gender ;
    …
  ]
```

The evaluation of classifier declarations results in the insertion of classification slots into an object. A classification slot will contain the mixin patterns of the mixins applied to the object that contains the slot. Each mixin application must check and update classification slots for possible conflicts. In an actual extension of the framework this can be realised by an ingenious system of delegating classifier information. We will not go into the details.

Other useful classifiers have been investigated. A covering classifier for example can be used to enforce the application of at least one mixin out of a selection of mixins. An object that is not complete with respect to a covering classifier is an abstract object.

4.7 Conclusion

In this chapter we discussed how our framework can be refined with an inheritance mechanism by adding a layer to it.

For this purpose we discussed the design issues that are involved, including issues such as multiple inheritance, constraining inheritance hierarchies, scoping issues, ... We came up with a novel inheritance mechanism on objects called mixin-methods, and discussed a language (Agora) that incorporated this inheritance mechanism. Other salient features of Agora are: it has a minimal syntax — essentially message passing syntax — due to its extensive use of reifier expressions; it is slot based; it features nesting of mixin-methods; it is prototype based.

We discussed a framework layer that handles mixin-method based inheritance. We showed that this layer is a refinement of our basic framework due to the fact that the mixin-method inheritance mechanism can be totally encapsulated, i.e. it has no effect on the interface of meta-objects. We discussed an internal object representation that can be used to implement mixin-methods. We also discussed cloning (or copying) mechanisms for objects.

Finally we showed how the refined framework can be used to define a set of useful extensions to Agora.