Chapter 5

A Reflective Framework

5.1 Introduction

Now that we have defined an open design for object-based and object-oriented programming languages, we can focus on how to turn this open design into a reflective system. Turning an open design into a reflective system differs in a fundamental way from the conventional approach of defining reflective languages.

Conventionally a reflective language is defined by giving a meta-circular interpreter for it. This meta-circular interpreter can contain various circularities that are resolved by imagining an entire tower of meta-circular interpreters. In an actual running system various mechanism are used to implement this tower. Resolving these circularities is an essential step in making a running reflective language.

We will take another approach. As we said in the chapter on reflection a programming language is turned into a reflective one by *extending* it with reflection operators. We will consider those open designs that are powerful enough so that reflection can be added as a full-fledged specialisation of the open design. The advantage is that in that case a formal relation exists between a reflective language and its open design. Reflection is added as an orthogonal language concept. Among others this means that the open design itself (meta-circularly defined or not) does not need to be altered in an ad hoc fashion to turn it into a running reflective system.

As discussed in the chapter on reflection, turning a system with an open design in a reflective system is a matter of 1) achieving a symbiosis of the implementation language of the open implementation and the engendered language, and 2) providing the necessary reflection operators that may or may not avoid reflective overlap. Our discussion will follow these steps.

A symbiosis between two object-oriented languages enables objects to freely travel from one language to the other. First we will show how an object-oriented language can achieve a symbiosis with its underlying object-oriented implementation language. We will also show that this can be done with a fairly general mechanism. A symbiosis between an object-oriented programming language and its implementation language will be achieved by the introduction of *conversion-methods* and objects that incorporate *reflection equations*. The properties of these will be discussed.

In practice, the choice of the reflection operators is an important issue. Reflection operators must give access to both the base- and meta-level interface of the meta-system. A fully reflective language must give access to the entire base- and meta-level interfaces. We will discuss different sets of operators each with different characteristics, and show that to a certain degree, making a choice between them is a matter of taste.

Exactly as discussed, both the linguistic symbiosis and the reflection operators are added to the framework as an extra layer, i.e. reflection operators are in some sense not different of any other extension of the framework that adds new sorts of expressions and new sorts of objects. In fact the extension of the framework with reflection involves extending the framework's object hierarchy (with conversion objects that realise the symbiosis) and the expression hierarchy (with the actual reflection operators).

5.2 Object-based Reflection

5.2.1 Linguistic Symbiosis

Both Agora and its implementation language are object-oriented languages. The purpose of this section is to show how objects from Agora's implementation language can be used as Agora objects, i.e. how messages, expressed in Agora, can be sent to implementation language objects. Vice versa, we will show how Agora objects can be used as objects from the implementation language, i.e. how messages, expressed in the implementation language, can be sent to Agora objects. This is depicted informally in the following figure.

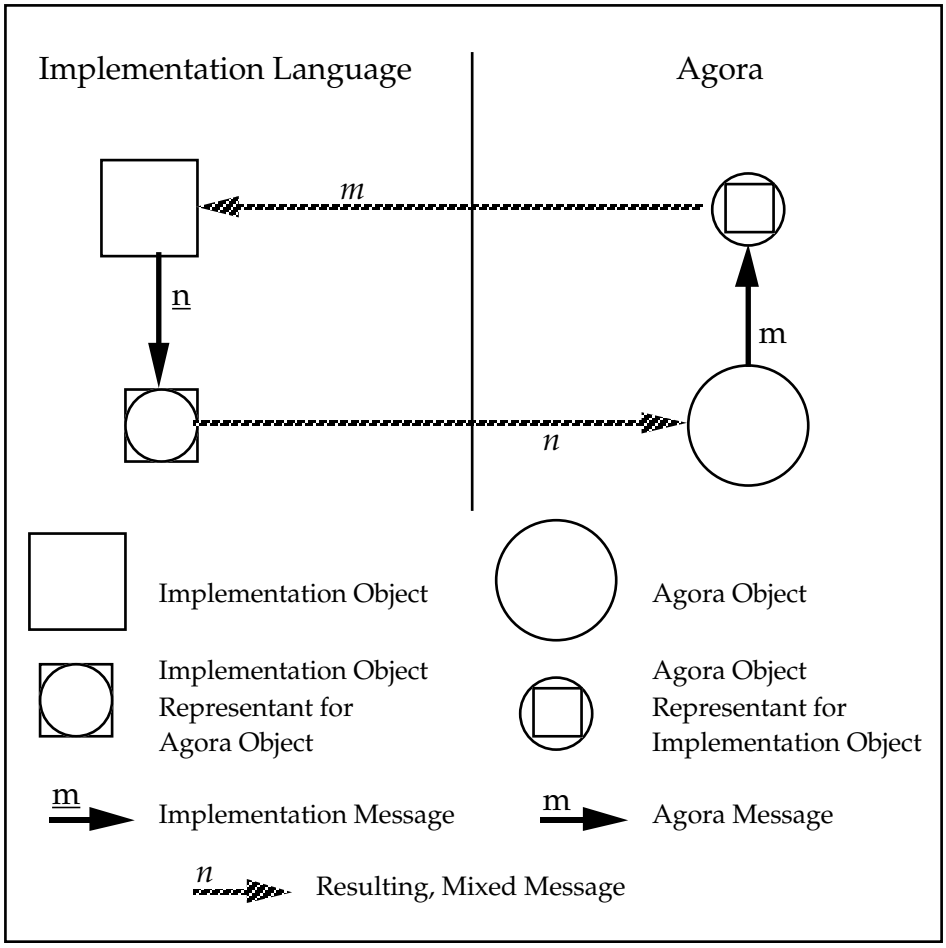

Figure 5.1

Before plunging into the technical details of the symbiosis of Agora and its implementation language, we will need some terminology. The distinction between Agora objects and implementation level objects will be blurred because after the symbiosis, objects will be able to travel between Agora and its implementation language. The simple terminological difference between Agora objects and implementation level objects is not good enough anymore. Therefor we will need a new terminology. The point is that we will need to make a distinction between the language in which an object is expressed and the language from which messages can be sent to an object. First of all we can make a distinction between *implicit messages* — messages expressed in the implementation language — and *explicit messages* — messages expressed in Agora. Secondly we will talk about an *explicitly encoded object* when this object is expressed in Agora, and about an *implicitly encoded object* when this object is expressed in the implementation language. Not every explicitly encoded object need to be referenced from within an Agora program. An object that can be sent implicit messages is called an *implicitly referable object*, an object that can be sent explicit messages is called an *explicitly referable object*. Finally we will simply talk about an implicit (explicit) object when this object is both implicitly (explicitly) encoded and referable. The following table summarises our terminology.

	Implementation Language	Agora
referable	Implicitly Referable Object	Explicitly Referable Object
encoded	Implicitly Encoded Object	Explicitly Encoded Object
referable & encoded	Implicit Object	Explicit Object

Figure 5.2

For one particular kind of objects this terminology can be interpreted in an ambiguous way. This is the source of much terminological confusion in object-oriented reflective programming languages. A meta-object is an object that is both implicitly and explicitly referable, albeit with two different protocols. To illustrate this, let us have a look at how explicit objects are represented in the implementation language. Each explicit object is represented at the implementation level by an implicitly referable meta-object. The latter will be called the *representation* of the former, the former will be called the *referent* of the latter. An explicit message to an explicit object is represented (or implemented) by an implicit message to the implicitly referable representation of that object, albeit a message with a different signature.

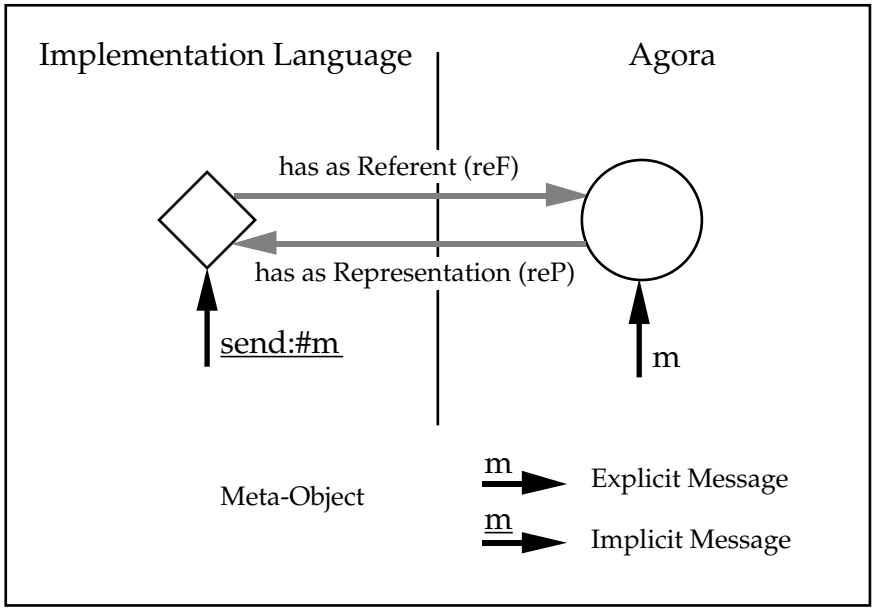

Figure 5.3

If the relation '`reP`' associates each explicit object with its representation object, and the relation 'reF' associates each meta-object with its referent object, then the following holds for message passing between objects (depicted in the next table). Pattern objects are conveniently represented as '#x:y:z:', and argument lists as '{a1, … an}'.

Agora Objects and Their Representations (rule 1a)

```
reP[o x1:a1 x2:a2 … xn:an]
                    =
reP[o] send:#x1:x2:…xn:
       client:(StandardClient private:{reP[a1], … reP[an]})

if o, a1, a2, … , an are explicitly referable objects,
and o x1:a1 x2:a2 … xn:an is an explicitly sent message
```

Meta-Objects and Their Referents (rule 1b)

```
reP[ reF[mo] x1:reF[ma1] x2:reF[ma2] … xn:reF[man] ]
                    =
  mo  send:#x1:x2:…xn:
      client:(StandardClient private:{ma1 … man})

if mo, ma1, ma2, … , man are implicitly referable meta-objects,
and mo send:(…) client:(…) is an implicitly sent message
```

Equality of Referents and Representations (rule 1c)

$$reF[mo1] = reF[mo2] \Leftrightarrow mo1 = mo2$$

$$reP[o1] = reP[o2] \Leftrightarrow o1 = o2$$

if mo1, mo2 are implicitly referable meta-objects, and
if o1, o2 are explicitly referable objects

Notice that it can be proved that the relations reF and reP are in a sense inverse relations with respect to message passing. It can be shown that an explicit message sent to 'reF[reP[o]]' has the same effect as an explicit message sent to 'o'.

```
reP[ reF[reP[o]] x1:reF[reP[a1]] … xn:reF[reP[an]] ]
=                                                          (rule 1b)
reP[o] send:#x1:x2:…xn:
       client:(StandardClient private:{rep[a1] … reP[an]})
=                                                          (rule 1a)
reP[o x1:a1 … xn:an]

therefore                                                  (rule 1c)
reF[reP[o]] x1:reF[reP[a1]] … xn:reF[reP[an]] = o x1:a1 … xn:an
```

Conversely it can be shown that an implicit 'send:client:' message to 'reP[reF[mo]]' has the same effect as an implicit 'send:client:' message to 'mo'.

```
reP[reF[mo]] send:#x1:x2:…xn:
             client:(StandardClient private:
                        {reP[reF[ma1]], … , reP[reF[an]]})
=                                                          (rule 1a)
reP[ reF[mo] x1:reF[ma1] … xn:reF[man] ]
=                                                          (rule 1b)
mo  send:#x1:x2:…xn:
    client:(StandardClient private:{ma1 … man})
```

Returning to the question of how to construct a symbiosis, we can now consider two conversion methods for implicitly referable objects. The first — named 'asImplicit' — turns a meta-object into an implicitly referable object with the protocol of the referent of the meta-object. The second — named 'asExplicit' — turns an arbitrary implicitly referable object into a meta-object, i.e. into a representation of an explicitly referable object with the same protocol as the initial implicitly referable object. Note that these conversion methods are implementation level methods, i.e. they can only be sent at the implementation

level. The two conversion methods are illustrated in the following figure.

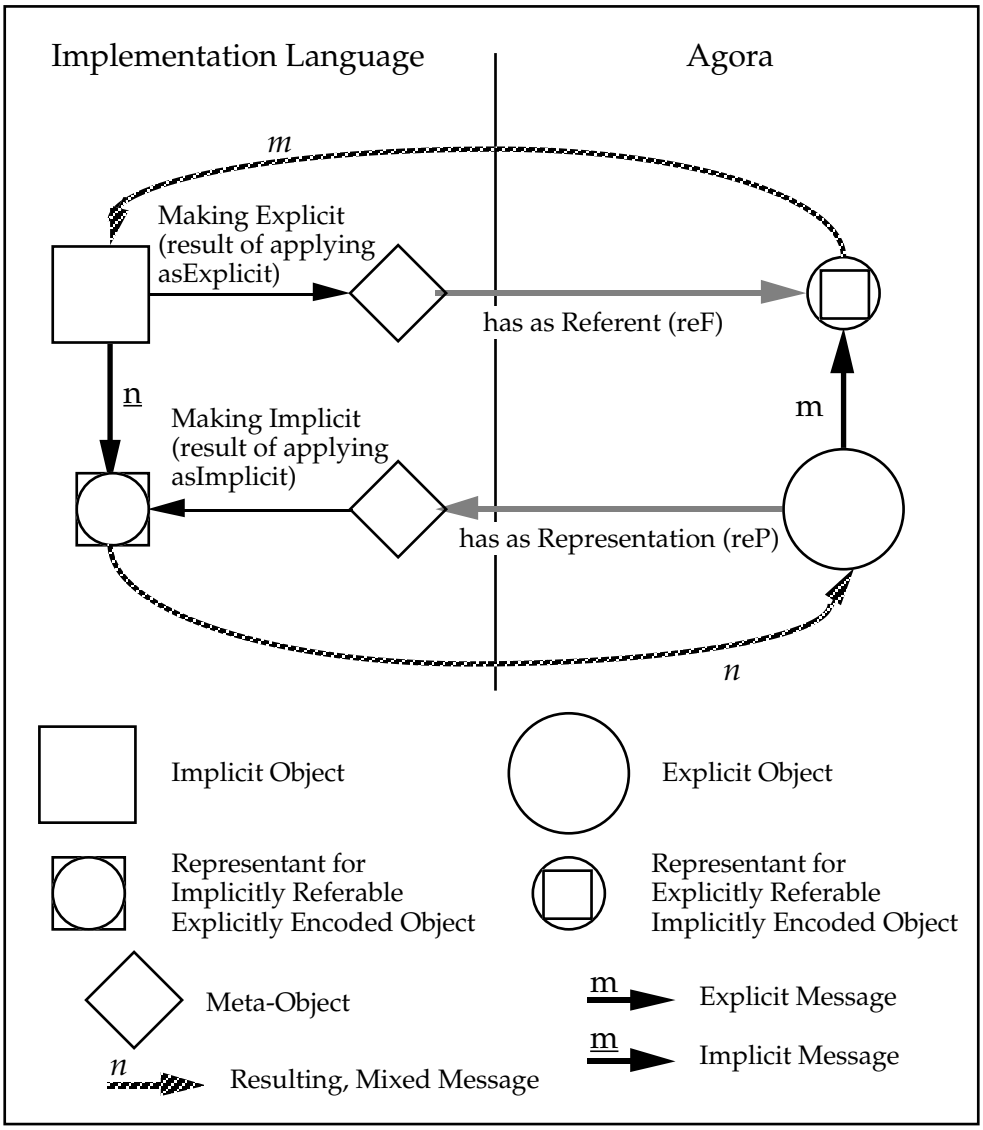

Figure 5.4

These conversion methods are crucial in achieving a symbiosis between Agora and its implementation language. The 'asImplicit' conversion method allows an object to travel from an Agora program to a program expressed in the implementation language. Conversely the 'asExplicit' conversion allows an object to travel from the implementation level to an Agora program. The conversion methods are defined by the following equalities :

Reflection Equations for Conversion Methods

reP[o *x1*:reF[a1 asExplicit] ... *xn*:reF[an asExplicit]] asImplicit
=
(reP[o] asImplicit) *x1*:a1 ... *xn*:an

if o is an explicitly referable object
a1, a2, ... are implicitly referable objects
and o *x1*:... ... *xn*:... is an explicitly sent message

(reF[o asExplicit]) *x1*:a1 ... *xn*:an
=
reF[(o *x1*:(reP[a1] asImplicit) ... *xn*:(reP[an] asImplicit)) asExplicit]

if o is an implicitly referable object
a1, a2, ... are explicitly referable objects
and o *x1*:... ... *xn*:... is an implicitly sent message

(o asExplicit asImplicit) = o = (o asImplicit asExplicit)

if o is an implicitly referable meta-object

The 'asImplicit' and 'asExplicit' conversion methods are by axiom inverse methods for meta-objects. Intuitively, the previous equalities can be interpreted as a form of distribution of the 'asImplicit' and 'asExplicit' conversions over message passing.

The 'reP' and 'reF' relations on the one hand and the 'asImplicit' and 'asExplicit' conversion methods on the other hand should not be confused. The former are relations between implicit objects and explicit objects, i.e. a relation that can be observed to exist, or not. The latter are methods that must be explicitly applied. Furthermore the protocol changes involved are of a different nature. The 'reP' relation, for example, relates an explicitly referable object with an arbitrary protocol to an implicitly referable object with a protocol that is comprised of essentially a 'send:client:' method. The 'asImplicit' conversion method, however, converts the representation of an explicitly referable object with an arbitrary protocol to an implicitly referable object with the same protocol.

The role of the 'asImplicit' and the 'asExplicit' conversion methods will be illustrated with an example. Consider implicit expression objects. The method '`asExplicit`' for expressions converts an implicitly encoded expression object (i.e. an implementation object of type '`AbstractExpression`') into an explicitly referable object. The resulting object can receive explicit evaluation messages. Upon reception of an evaluation message '`eval:`', expressed in Agora's message passing, the converted object translates this message to an '`eval:`' message on the implementation level. It also takes care that the context argument is translated into an implicit referable object, and the result is translated back into an explicitly referable object. These translations are necessary since the message was sent from within an Agora program. Obviously the 'asExplicit' conversion method will play an important role in making primitive objects — that are present in the implementation — available to Agora programs.

Conversely, an Agora object that implements an '`eval:`' method, can be transformed to a implicitly referable object with the conversion method '`asImplicit`'. This method will translate this explicitly encoded object into an implicitly referable expression object, i.e. the implementation-language-representant (preferably of type '`AbstractExpression`') for explicitly encoded expression objects that can receive implicit evaluation messages. Upon reception of an implicit evaluation message '`eval:`' the converted object translates this

message to an Agora style 'eval:' message. Care is taken that the context argument is translated to an Agora object, and the result is translated back to an implementation object. Obviously, the 'asImplicit' conversion method will play an important role in the implementation of reflective operators.

So, we see that this form of symbiosis is but a mere extension of handling primitive data-types, as can be found in most implementations of programming languages. A primitive data-type is a data-type from the implementation language that is transported to the implemented language. In most languages, only the direction of implementation language to implemented language is supported; from the viewpoint of reflection, the other direction is much more interesting. In reflection terms the conversion methods allow objects to "shift levels".

The implementation of conversion methods for the symbiosis is straightforward in principle, but tedious in practice. Let us first look at the 'asExplicit' method. This method is for example defined for expression objects. It converts an implicit expression object into an explicitly referable expression object of which the definition is found below. Notice that the class of explicitly referable expression objects is a concretisation of the abstract class of meta-objects. Its implementation is one of translating explicit messages to implicit messages according to the following schema.

asExplicit Conversion Method

```
class AbstractExpression extends ObjectThatCanBeMadeExplicit
  abstract class attributes
   ExplicitlyReferableExpression
  methods
   concrete asExplicit result AbstractMetaObject
     ^ExplicitlyReferableExpression expression:self
endclass
```

Explicitly Referable Expressions

```
class ExplicitlyReferableExpression extends AbstractMetaObject
  constants
   EvalPattern = KeywordPattern name:"eval:"
  instance variables
   expression:AbstractExpression
  methods
   concrete send:pattern client:client result AbstractMetaObject
     if pattern = EvalPattern
       then
         ^(expression eval:(client arguments asImplicit))asExplicit
       else
         … raise an error
endclass
```

The implementation of the reverse 'asImplicit' conversion method is as straightforward as the previous one. This conversion method is defined only on meta-objects. In principle it translates an anonymous meta-object into an anonymous implementation level object. More specific variants of this conversion method can be useful. For example an 'asImplicitExpression' conversion method would translate a meta-object into an object of the 'ExplicitlyEncodedExpression' class of which the definition can be found below. Again, this class does a simple translation of messages.

asImplicit Conversion Method

```
class AbstractMetaObject
  abstract class attributes
   ExplicitlyEncodedExpression
  methods
   concrete asImplicitExpression result AbstractMetaObject
     ^ExplicitlyEncodedExpression object:self
endclass
```

Explicitly Encoded Expressions

```
class ExplicitlyEncodedExpression extends AbstractExpression
  constants
   EvalPattern = KeywordPattern name:"eval:"
  instance variables
   object:AbstractMetaObject
  methods
   concrete eval:context result AbstractMetaObject
     ^(object
         send:EvalPattern
         client:(StandardClient arguments:(context asExplicit))
       ) asImplicit
endclass
```

Finally, note that because meta-objects (not their referents) are implicitly referable objects, the 'asExplicit' conversion method should be defined for them.

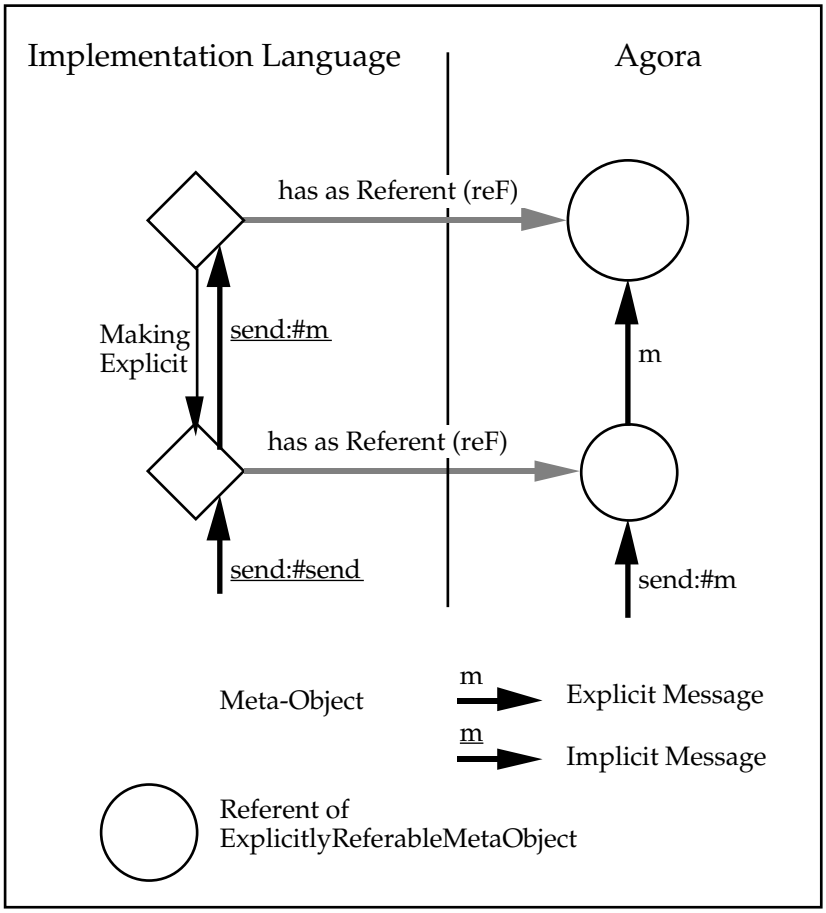

Figure 5.5

The according conversion class can be found below. It encodes meta-objects to which explicit 'send:client:' messages can be sent. Remark that, on meta-objects, the 'asExplicit' conversion can be applied an infinite number of times.

Explicitly Referable Meta-objects

```
class ExplicitlyReferableMetaObject extends AbstractMetaObject
  constants
   SendPattern = KeywordPattern name:"send:client:"
  instance variables
   aMeta:AbstractMetaObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     if pattern = SendPattern
       then
         ^(aMeta
             send:(client firstArgument asImplicit)
             client:(client scndArgument asImplicit))asExplicit
       else
         … raise an error
endclass
```

5.2.2 Simple Meta-Programming Operators for Agora

To illustrate the linguistic symbiosis we will discuss a set of reflection operators that is directly inspired by the above conversion methods. As we saw in the previous section the 'asImplicit' method can be used for example to convert an implicitly referable expression object into an explicitly referable expression object. This is called a quoting operator when provided as a language construct. Similarly, meta-objects can be converted into explicitly referable meta-objects. We will also illustrate how the inverse operations — that of converting an explicitly encoded expression or meta-object into an implicitly referable object — can be made useful.

The quoting reifier (form: 'e **quote**') allows us to get hold on expressions as Agora objects in what is usually called a *meta-program*. A quoted expression is an object that can be sent an explicit evaluation message, given a context as argument. The following meta-program evaluates the *object-level program* '"hello world" print' in an initially empty context. We presume that somewhere an appropriate prototype 'EmptyContext' has been defined. This prototype should conform to the protocol of standard contexts.

```
EmptyContext define: … ;     —— an initially empty context

aProgram define ;
aProgram  <- ("hello world" print)quote ;
aProgram eval:(EmptyContext clone)  —— prints "hello world"
```

This is a typical example of *meta-programming*: allowing us to manipulate programs as first-class objects, but on the other hand absorbing (leaving implicit) the evaluator for these programs. Some remarks must be made. Consider the following example. The object-level program creates a point object that is returned as result[1].

[1] Note that in Agora block expressions do *not* evaluate to something like closures (such as is the case in Smalltalk) but rather all component expressions are evaluated. The return reifier indicates what result must be returned.

```
aProgram <- ( [ Point define: … ;   —— a point prototype
                Point x:3 y:4 return ] )quote ;

p <- aProgram eval:(EmptyContext clone) ;

p x   —— ERROR: does not understand
      —— p is a result at the meta-level !
      —— therefore p is a meta-object
```

The first remark is that due to the fact that the evaluation is done in an explicitly given context the object-level program in the example can not refer to any of the prototypes defined in the meta-level program. Object-level programs must be 'self-contained' with respect to the referenced objects. Secondly, and more importantly, it must be noted that the result of an explicit evaluation is a meta-object. This is not only a direct result from the definition of our conversion methods, but it is also what we want. Whereas an object-level program deals with referents directly, the meta-program deals with the representations (meta-objects) of the objects of its object-level program. An evaluator (or a meta-system in general) that does not respect this is said to be a level-crossing evaluator [Smith82].

The implementation of the quoting operator is straightforward, and relies on the symbiosis of Agora and its implementation language. A quote reifier returns, upon evaluation, its receiver as an explicitly referable Agora object.

Quoting Expressions (without precautions to avoid reflective overlap)

```
class AbstractExpression
  methods
   reifier quote result AbstractMetaObject
         using (context:StandardContext)
     ^(self asExplicit)
endclass
```

Explicit meta-objects can be obtained in a way that is similar to the way expression objects are obtained. Similar to the quoting operator we introduce a reifier (form: '`e` **asMeta**') that transforms the representation of its evaluated receiver into an explicitly referable meta-object. Here again a typical example of a meta-program can be given. A meta-program that sends an explicit message to a meta-object. We presume that somewhere appropriate prototypes 'UnaryPattern' and 'EmptyClient' have been defined. These prototypes should conform to the protocols of respectively standard patterns and standard clients.

```
UnaryPattern define: … ;     —— a pattern prototype
EmptyClient define: … ;      —— a client prototype
Point define: … ;            —— a point prototype

metaOfP define ;
p define: Point x:3 y:4;
metaOfP  <- (p asMeta) ;
result <- metaOfP send:(UnaryPattern name:"x")
                  client:(EmptyClient clone)
```

Similarly to the above example, and for the same reasons, the result of an explicit message to a explicitly referable meta-object is a meta-object.

The definition of this new operator is as straightforward as the definition of the quote operator. It also relies on the symbiosis of Agora and its implementation language.

Accessing Meta-Objects

```
class AbstractExpression
  methods
   reifier asMeta result AbstractMetaObject
         using (context:StandardContext)
     ^((self eval:context) asExplicit)
endclass
```

Notice that, in contrast with other object - meta-object approaches, an object does not contain a reference to its meta-object but, rather, that object and meta-object are different views for communicating with an object (much in the style of the reP relation). Every object in Agora is implemented as an object in the implementation language (typically called the meta-object, and having a 'send:client:' method in its protocol). According to our symbiosis this latter object can be made explicit, via the 'asExplicit' conversion method. It is made explicit as an Agora object, such that Agora messages can be sent to it. The kind of messages that can be sent are 'send:client:' messages. The implementation level objects that implement Agora objects, are in that respect no different of, say, context-, or expression objects at the implementation level. Still, the fact that there is already a relationship (i.e. the reP and reF relation) between Agora objects, and implementation level meta-objects (which is not the case for context, or expression objects for example) can make this a bit confusing.

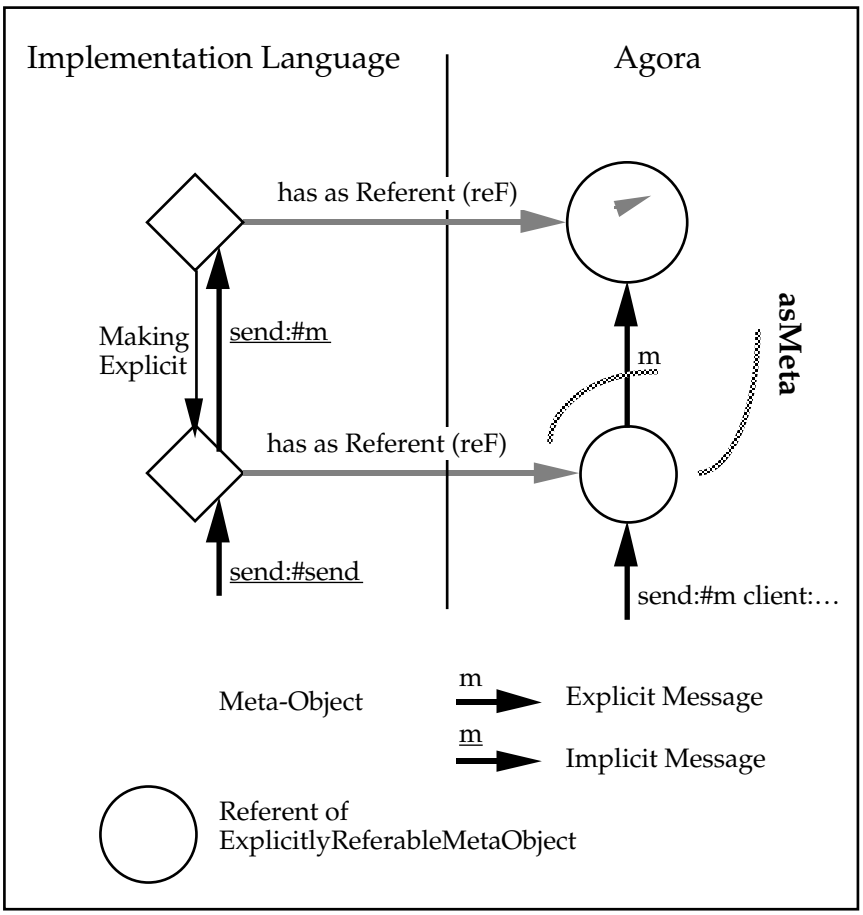

Figure 5.6

5.2.3 Simple Reflection Operators

The most important aspect of the meta-level interface of the open implementation of Agora is the extension of the class hierarchy of expressions and the class hierarchy of objects. In this section we will discuss two reflection operators that make this aspect of the meta-level interface available to Agora programs. These operators, also, are directly inspired by the conversion methods 'asImplicit' and 'asExplicit'. Whereas in the previous section we made use of the fact that expression and meta-objects can be made explicitly referable, we will now make use of the fact that explicitly encoded expression and meta-objects can be made implicitly referable by the 'asImplicit' conversion method.

The introduction of new sorts of expressions in an open implemented programming language, in principle, goes hand in hand with the introduction of new syntactic constructs. Mere extension of the expression class hierarchy is not enough, it is not even the goal. The goal is to be able to construct program trees that make use of the newly added expression objects. As we saw before, this can be realised, for example, with a generic syntax.

To keep things simple, however, the construction for reflectively adding new types of expressions, discussed here, will be of a flavour that avoids this complication. This construction is reminiscent of reifier functions in e.g. 3-Lisp [Smith82], albeit of a more primitive nature.

Furthermore it is our intention to illustrate the dynamic character of using the meta-level interface that comes with reflection. Previously, in the 'plain' open implementation, the usage of base- and meta-level interface were strictly separated in time. In case of reflection this need not be so.

The point is to offer a reflection operator that allows the dynamic extension of the program tree with explicitly encoded expression objects, i.e. an operator that, given a Agora object that implements an evaluation method, virtually installs this object in the program tree. The reifier (form: 'e **asExpression**') we propose for this purpose is more or less the reverse of the quote operator. It evaluates its receiver expression 'e' — that will be called the expression-definition of the absorbed expression —, transforms the result — that will be called the explicit expression-object of the absorbed expression — to an implicitly referable expression object, and sends an implicit evaluation method to this transformed object. Whereas the quote operator reifies parts of the program tree into explicitly referable objects, the asExpression operator absorbs explicitly encoded expression objects into the program tree.

Consider the following example. The goal is to construct an expression type that reifies the current context. For this purpose an appropriate expression object is defined. Each time this explicit expression object is absorbed in the evaluation process, by means of the 'asExpression' reifier, it reifies the current context.

```
Point define: … ;              —— a point prototype

MakeCurrentContextExp Mixin:
    [ eval:context Method:[ context return ] ] ;
CurrentContextExp define: Object MakeCurrentContextExp ;

currentContext define;
currentContext <- (CurrentContextExp asExpression) ;

a <- ( Point x:3 y:4 )quote ;
p <- a eval:currentContext ;
(p asObject) x print                          —— prints 3
```

This example features two different forms of reflective overlap. Firstly, the context that is reified by the 'currentContextExp' expression, is both reified and left implicit. This is apparent in the fact that the variable that points to the reified context also is part of this reified context. Secondly, and more importantly, the evaluation method of the explicit expression object is evaluated in the same context that it reifies. This evaluation method has, for example, access to the point prototype, both directly and via its context argument. Whereas the first kind of reflective overlap is the result of how the above program is formulated, the second kind is a direct result of the definition of the 'asExpression' reifier. We will see in the next section how an alternative set of reflection operators that avoid reflective overlap can be defined.

Explicitly encoded meta-objects can also be made implicitly referable. The reifier 'asObject' allows the absorption of explicitly encoded meta-objects. In the following example meta-objects are constructed that reply lazily to messages. The result of a message sent to a lazy object is computed only if a message is sent to this result. Therefore two different sorts of meta-objects are defined. The first kind ('Lazy' objects) that contains a reference to the object that is made lazy. The second kind ('ResultHolder' objects) that act as representants for the results of the messages sent to a lazy object. Notice that in the example, the first kind of meta-objects is put to use by an explicit application of the 'asObject' reifier. Whereas the second kind is created in the execution of an explicitly encoded 'send:client:' message. Since this latter is executed at the meta-level the so created meta-object is automatically absorbed.

```
MakeResultHolder Mixin:
  [ receiver define ; pattern define ; client define ;
    receiver:r pattern:p client:c CloningMethod:
      [ receiver <- r ; pattern <- p ; client <- c ] ;
    send:p client:c Method:
      [ (ResultHolder
            receiver: (receiver send:pattern client:client)
            pattern:p
            client:c) return ]
  ] ;
ResultHolder define: Object MakeResultHolder ;

MakeLazy Mixin:
  [ who define ;
    who:w CloningMethod: [ who <- w ] ;
    send:p client:c Method:
        [ (ResultHolder receiver:who pattern:p client:c) return ]
  ] ;
Lazy define: Object MakeLazy ;

Point define: … ;

p define: (Lazy who:((Point x:3 y:4) asMeta)) asObject ;
—— p contains a lazy point now
```

The definitions of both the 'asObject' and 'asExpression' operators are straightforward.

Installing Expressions and Meta-Objects (without precautions to avoid reflective overlap)

```
class AbstractExpression
  methods
   reifier asObject result AbstractMetaObject
         using (context:StandardContext)
     ^((self eval:context) asImplicit)
endclass
```

```
class AbstractExpression
  methods
   reifier asExpression result AbstractMetaObject
         using (context:StandardContext)
     ^(((self eval:context) asImplicit) eval:context)
endclass
```

This last definition is an interesting one since it illustrates two important aspects of a reflective system. The first is the notion of reflection levels. In the definition of the 'asExpression' reifier it is apparent that two evaluation messages are sent. Unlike a recursive call to the evaluator function, the above calls to the evaluator are 'cascaded'. The second call to the evaluator is sent to the result of the first evaluation. This obviously gives rise to layers of evaluation (reflection levels). All usages of the 'asExpression' reifier need two layers of evaluation. Closely connected is the notion of reflective overlap. The 'asExpression' reifier suffers from it since both layers of evaluation use the same context.

5.2.4 Nature of Meta-Programs and Reflective Overlap

As illustrated in the example in which contexts where reified, the 'asExpression' reifier introduces a form of reflective overlap. An explicitly encoded expression object is evaluated in the same context that will be passed as argument of the implicitly sent evaluation message. This reflective overlap can, but must not necessarily, be avoided. If we do want to avoid reflective overlap, the question of what should be considered part of the meta-program and what should be considered part of the object-level program must be answered.

As we saw in the first quoting example, quoting introduces a natural boundary between meta-program and object-level program. The result of a quote expression is a new object-level program. Conversely the expression-definition used in an 'asExpression' reifier should be part of the meta-program since, conceptually, it adds an expression type to the open implementation. Consider the following example. Obviously the main part of the program can be interpreted as a meta-level program. It defines a 'constant 3' expression, and evaluates some quoted object-level program. It is more than natural that, in the object-level program, this newly defined expression type can be used. Therefore the expression-definitions used in calls to the asExpression reifier from within the object-level program should be evaluated in the context of the meta-program rather than in the context of the object-level program. Such a strict separation of the meta-context and the object-level context also solves our problem of reflective overlap. The following example features the variant reifiers 'cleanQuote' and 'cleanAsExpression' that avoid reflective overlap.

```
MakeConstant3Exp Mixin:
    [ eval:context Method: [(3 asMeta) return ] ] ;
Constant3Exp define: Object MakeConstant3Exp ;

EmptyContext define: … ;     —— an initially empty context

aProgram define ;
aProgram <- ((Constant3Exp cleanAsExpression) print ) cleanQuote ;
aProgram eval:(EmptyContext clone)  —— prints "3"
```

Trivially, reflective overlap can be avoided by keeping track of meta-contexts by means of a stack mechanism. The explicitly given context in which a quoted expression is evaluated is pushed on the stack of meta-contexts. This stack is popped to return to the meta-level when the receiver expression of the asExpression reifier is evaluated.

Quoting Expressions

```
class QuoteExpression extends AbstractExpression
  instance variables
   quote: AbstractExpression
   inContext:StandardContext
  methods
   concrete eval:context result AbstractMetaObject
     ^(quote eval:(context push:inContext))
endclass
```

```
class AbstractExpression
  abstract class attributes
   QuoteExpression
  methods
   reifier cleanQuote result AbstractMetaObject
         using (context:StandardContext)
     ^((QuoteExpression quote:self inContext:context) asExplicit)
endclass
```

Installing Expressions and Meta-Objects

```
class AbstractExpression
  methods
   reifier cleanAsExpression result AbstractMetaObject
         using (context:StandardContext)
     ^(((self eval:(context pop)) asImplicit) eval:context)
endclass
```

5.2.5 Dynamic Reflection and Infinite Regress

As we said in the previous section, reflective overlap need not necessarily be avoided. One particular case where reflective overlap comes in handy is in the definition of *dynamic reflection*. As we explained in the second chapter, dynamic reflection is characterised by the fact that the number of times a program regresses is dynamically determined. Below is an example of a program that regresses infinitely (reflecting upon one's own behaviour in an infinitely recursive way). A regression expression is used in the evaluation of its own evaluation method. This definition looks very similar to a meta-circular definition, except for the fact that in this case there is no special provision to 'bottom out' of the circularity. The meta-circular definition is effectively used in its own interpretation.

```
MakeRegressingExpression Mixin:
  [ eval:context Method:
      [(RegressingExpression asExpression) return ] ] ;
RegressingExpression define: Object MakeRegressingExpression;

(RegressingExpression asExpression) —— infinite regression
```

The definition above is only possible in the case where the version of the 'asExpression' reifier is used that suffers from reflective overlap. Obviously, infinite meta-regress is easily constructed with such reifiers. Less obvious is how this sort of reflection can be applied to practical situations. This is reminiscent of what we said in our introduction of reflection: what use is it to keep on reasoning about one's self if this does not improve one's reasoning about the world. The use of dynamic reflection is in fact an open question. Here, we only point out a possible candidate that uses dynamic reflection in a useful way.

We talk about dynamic reflection when the number of 'reflection levels' is dynamically determined. One particular form of dynamic reflection occurs when the number of explicit meta-objects a particular object has, is dynamically determined — i.e. if the 'asObject' reifier has been applied a dynamically determined number of times. Notice that this is another kind of dynamic reflection than the above reifier that uses itself recursively in its evaluation method (leading to an undetermined number of evaluation levels). The dynamic aspect here has to do with levels of explicitly encoded 'send:client:' methods that are used in sending 'send:client:' messages (as depicted if figure 5.6). We will try to show that this can occur in a practical situation.

Consider writing a meta-circular definition for the Agora framework. One part of the job in doing so, is implementing a linguistic symbiosis between the newly defined Agora and its implementation language, the already defined Agora. The fact that both sorts of objects — the implicit Agora objects, and the explicit Agora objects — are so closely related doesn't seem to help. The problem is that objects need to shift levels. But, this is exactly the functionality provided by the 'asObject' and 'asMeta' reifiers. So, in the reflective variant of Agora, level shifting of objects can be absorbed. This is illustrated in the following sample of a hypothetical meta-circular Agora definition.

```
MakeMetaObject Mixin:
  [ …

    send:pattern client:client Method: [ … ] ;

    asImplicit Method: [ (self asObject) return ]

    … ] ;

MakeObjectThatCanBeMadeExplicit Mixin:
  [ asExplicit Method: [ (self asMeta) return ] ]
```

Notice that with each application of the 'asObject' reifier (to some object) in a program executed by the meta-circular Agora interpreter corresponds an application of the 'asObject' reifier in the code of the meta-circular interpreter (in fact to the meta-object, see figure 5.7). The number of times the 'asObject' reifier is applied to some object in the meta-circular interpreter is determined by the program it is evaluating. From the standpoint of this meta-circular definition this number is dynamically determined.

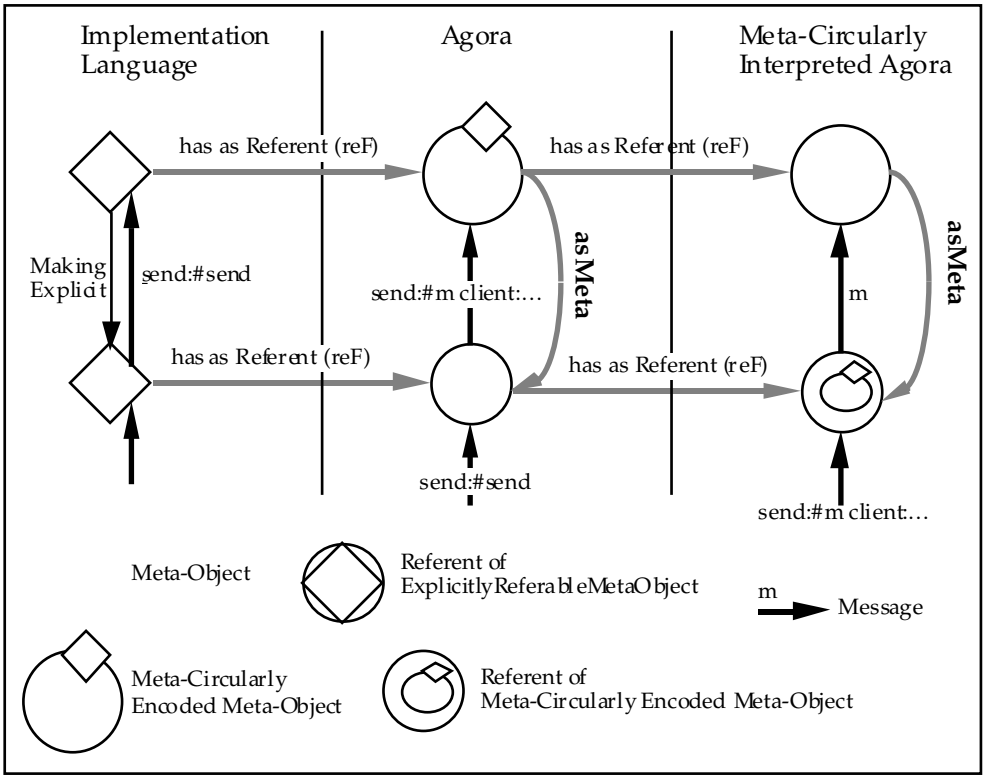

Figure 5.7

5.2.6 Full Abstraction and Compositionality

A final issue that is partially left open, is the role of compositionality and full abstraction in reflective programming languages. Apart from extensibility issues (as discussed before), it is clear that both concepts have an important role to play.

First of all full abstraction guarantees us that the meta-level programmer can not 'mess things up' more than is possible at the object level. For example, in a reflective object-oriented programming language where objects are not represented fully abstractly, a programmer can always break the encapsulation of objects. Consider again the non-abstract representation of meta-objects in the form of slot objects (see chapter 3). Imagine a reflective programming variant of Agora based on this alternative kind of meta-objects. As shown in the next example, in that case encapsulation of objects can not be ensured. The example features a turtle class that tries to encapsulate its location and heading variables. However, at the meta-level these instance variables can be freely accessed.

```
UnaryPattern define: … ;     —— a pattern prototype

MakeTurtle Mixin:
  [ location define: Point rho:0 theta:0*pi ;
    heading define: 0*pi ;
    home CloningMethod:[ … ];
    turn:turn Method: [ … ] ;
    forward:distance Method: [ … ] ] ;
Turtle define: Object MakeTurtle ;

turtleLocation define ;
aTurtle define: Turtle home ;
```

```
-- breaking the turtle's encapsulation
turtleLocation <-
  (((aTurtle asMeta)
       lookup:(UnaryPattern name:"location")) value) asObject
```

Another way to look at this is that in a non abstract implementation, the view one has on objects at the implementation level is a finer view (finer in the sense that one sees more implementation details) than one has at the programming level. Since in a reflective system it is possible to switch between these levels, it is always possible to take the finer view. This makes reasoning about programs more difficult, and diminishes reusability. Similar remarks apply for compositionality.

Compositionality and full abstraction also have implications on possible optimisation techniques. It is for example, easier to provide alternative implementation strategies for abstract object representations than for non abstract object representations. Also reification of expressions in a compositional way is a prerequisite for a compositional semantic definition of a reflective language [Malmkjær90].

So, compositionality and abstraction, indeed play an important role in reflective programming languages that transcends the role they play in the definition of open implementations. However, it is not yet possible to estimate the full consequences of both.

5.3 Object-Oriented Reflection

The above reflection operators do not give full access to the open implementation of Agora. Although an operator was presented to add new expression types one important aspect of Agora was ignored: that of reifiers. Another element that was ignored is the ability to extend the existing classes from the framework that constitutes Agora's implementation. We will show that a more fine-grained linguistic symbiosis is needed.

5.3.1. The Evaluation and Declaration of Reifiers

Since Agora itself is a full-fledged object-oriented programming language, considerable freedom exists in the choice of reflection operators. Moreover, since Agora aims to be a general purpose programming language, an important factor in the choice of reflection operators, apart from being complete, is their practical applicability, and ease of use.

Agora is best extended with new expression types by the addition of new reifiers. We will discuss the different characteristics of two sorts of reifiers that exist in Agora: that of reifier classes, and that of reifier methods. We will see that reifier classes are more appropriate for dynamic reflection and that reifier methods are more appropriate for static reflection. The evaluation of reifier expressions (both messages and receiverless messages) has been left open until now. All that was said is that somehow each time a reifier expression is evaluated, a corresponding reifier method or class has to be evaluated. Given the above symbiosis, we are now ready to show a possible interpretation of reifier expressions.

Reifier Classes
The first kind of reifier expressions that will be discussed are the receiverless reifier expressions, or also called reifier pattern expressions. An example reifier pattern expression is given below. The receiverless 'trace' reifier will first give rise to the creation of an implicit trace expression object that is then sent an implicit evaluation message.

```
p define: Point ;
p x:3 y:4 ;

p <- trace:p ;
—— from here on all messages to p will be traced
```

We are now ready to show how reifier pattern expressions can be interpreted. Two choices exist according to whether we want to avoid reflective overlap or not. We will take the latter choice. The idea is that the expression object that corresponds to a reifier pattern is to be found in the evaluation context. The corresponding expression object is looked up by sending a message to the evaluation context. The arguments of this message are explicit expression objects. When a correct expression object is found, it is made implicit and it is sent an evaluation message. Notice that the pattern with which the expression object is looked up is a reifier pattern.

Agora Receiverless Reifier Message Passing

```
class ReifierPatternExpression extends AbstractExpression
  abstract class attributes
   StandardClient
  instance variables
   pattern:AbstractReifierPattern
  methods
   concrete eval:context result AbstractMetaObject
     local variables
       reifierArguments:ArgumentList
       reifierInstance:AbstractMetaObject
     for each argument in pattern do
        reifierArguments add:(argument asExplicit)

     reifierInstance :=
         context private
           send:(pattern asCategory:context)
           client:(StandardClient arguments:reifierArguments)
     ^reifierInstance asImplicit eval:context
endclass
```

The following example shows how to declare reifier expression objects. The definition of a reifier object is that of any explicitly encoded expression object. In this case a tracing expression is implemented that defines how to make a tracing meta-object. The mapping between the '**trace**:' reifier pattern and the trace expression object takes the form of a private method declaration. This private method is executed each time the trace reifier is evaluated. Notice that it is declared with a reifier pattern as head.

```
MakeTracingObject Mixin:
  [ who define ;
    who:w CloningMethod: [who <- w ] ;
    send:pattern client:client
      method:
        [ … put trace information on screen …
          (who send:pattern client:client) return
        ] ] ;
TracingObject define: Object MakeTracingObject ;
```

```
MakeTracingExpression Mixin:
    [  exp define ;
       trace:e CloningMethod: [exp <- e ] ;
       eval:context Method:
          [ (TracingObject who:(e eval:context)) return ] ];
TracingExpression define: Object MakeTracingExpression ;

(trace:e) privateMethod: TracingExpression trace:e ;

p define: Point ;
p x:3 y:4 ;
p <- trace:p ;
—— from here on all messages to p will be traced
```

With the above mechanism local extensions to Agora can be made at run-time. A program is evaluated under a local extension of the class hierarchy. Such a local extension takes the form of a set of reifier declarations.

Other such mechanisms can be devised (e.g. global reifier declarations, recursive reifier declarations). They all share the property that programs can, during execution time, extend the set of reifiers that can be used. Obviously, in some cases this ability must go hand in hand with a mechanism for handling reflective overlap. The notion of meta-contexts can be reused for these purposes.

Reifier Methods

The evaluation of reifier expressions has been left open until now. All that was said is that somehow each time a reifier message is sent, the correct reifier method has to be executed. We are now ready to show how reifier messages can be interpreted as a special kind of messages. What is needed for this special interpretation is the notion of a linguistic symbiosis. Reifier messages are nothing but messages sent to converted expression objects. Besides passing all converted component expressions as arguments, the context also needs to be passed to the receiver expression object.

Agora Reifier Message Passing

```
class ReifierMessageExpression extends AbstractExpression
  abstract class attributes
   ReifierClient
  instance variables
   receiver:AbstractExpression,
   pattern:AbstractReifierPattern
  methods
   concrete eval:context result AbstractMetaObject
     local variables reifierArguments:ArgumentList
     for each argument in pattern do
        reifierArgumentsadd:(argument asExplicit)
     reifierArguments add:(context asExplicit)

     ^(receiver asExplicit)
         send:(pattern asCategory:context)
         client:(ReifierClient arguments:reifierArguments)
endclass
```

Thus, reifier declarations in Agora are nothing but special method declarations within expression objects. What differentiates a reifier from any other method is that it has an implicit context argument[2]. An example reifier declaration can be found below. It should be declared in a pattern class.

[2] In fact message passing to explicitly referable objects must be adapted accordingly. Also the client that carries the reifier arguments must allow one extra argument.

```
(PrivateMethod:righthand)
    using:context
    reifier:
     [ context privateSlots add: (MethodSlot key:self value:righthand) ]
```

The implementation of the "**using**:**reifier**:" reifier simply adds a reifier slot to the public part of the object in which this declaration took place. A reifier slot differs from other method slots by the fact that it can handle hidden context arguments (the formal argument name of the hidden argument is stored in this slot).

```
class AbstractReifierPattern
  abstract class attributes
   ReifierSlot
  methods
   reifier using:contextPattern reifier:body
       result AbstractMetaObject
       using (declarationContext:StandardContext)
     declarationContext publicSlots add:
       (ReifierSlot key:self value:body using:contextPattern)
endclass
```

The disadvantage of reifier methods is that they necessarily lead to a more static form of reflection. Reifiers can be declared in newly created expression classes. The problem is that when a program is being executed all expression-objects are already instantiated. One either has to devise some mechanism whereby reifier methods can be added to already existing expression objects — for example in the form of some reclassification mechanism — or one falls back upon a more static form of reflection. Let us consider the latter.

Expression classes must be defined prior to using them in the construction of program trees. This leads us to a form of reflection highly inspired upon the pragmatics of the existing open implementation of Agora. In practice the open implementation of Agora is used as follows. First, new kinds of expressions, objects, etc. are constructed by inheriting from, and extending the existing class hierarchies. Then, these new classes are used by the programming environment (e.g. the parser) to construct a program tree. Finally this program tree is executed. The difference is that now expression classes can be expressed in Agora.

The advantage of the above approach is that all the power that comes with nested mixins can be used in structuring the class hierarchy of for example expressions. Particular extensions of Agora can be grouped together in mixins, and nesting and overriding of mixins can be used to record dependencies between such groups of extensions.

The disadvantage of this approach is that, if we literally follow the above, the entire programming environment for Agora must be made explicit in Agora. In some cases this may be just what we wanted (the Smalltalk programming environment for example is explicitly encoded in Smalltalk). In other cases this may be a dramatic overhead. Other solutions exist however. The techniques used in opening up the implementation of Agora's evaluator may equally well be applied to a programming environment. It is then only a question of making the interface of this open implemented programming environment available to the Agora programmer. For the time being, this issue remains open.

Another, more important disadvantage is that reflection, in the above, is reduced to a static mechanism. Although in a literal sense programs implicitly reflect during run-time, all extensions to Agora are made prior to running a program with this extended version of Agora.

5.3.1. Need for a More Fine-Grained Linguistic Symbiosis

The above reflection operators do not give full access to the open implementation of Agora. What is lacking is the ability to extend existing classes from the framework that constitutes Agora's implementation. Agora's implementation hierarchy must be made accessible from within Agora. In practice this class hierarchy can be made available as a library of mixins. For the expression hierarchy, for example, this means that the following library of nested mixins — of which the root mixins are applicable to the root object — are made available:

```
MakeAbstractExpression
  MakeMessageExpression
  MakeReifierMessageExpression
  MakeAggregateExpression
  MakeLiteralExpression
  MakePatternExpression
  MakeReifierPatternExpression
MakeAbstractPattern
  MakeUnaryPattern
  MakeOperatorPattern
  MakeKeywordPattern
MakeAbstractReifierPattern
  MakeUnaryReifierPattern
  MakeOperatorReifierPattern
  MakeKeywordReifierPattern
```

Similar libraries of mixins must be made available for all other class hierarchies in the implementation. Each of these mixins can then be used to extend the implementation hierarchy of Agora.

The symbiosis of Agora and its implementation language that was achieved in the previous section is not sufficient for this purpose. Given our goal, a more fine-grained symbiosis of Agora and its implementation language is called for. In particular, what is needed is that Agora objects can inherit from implicitly encoded objects. If we divide an object into sub-objects corresponding to the inheritance structure, then the following figure depicts what is needed.

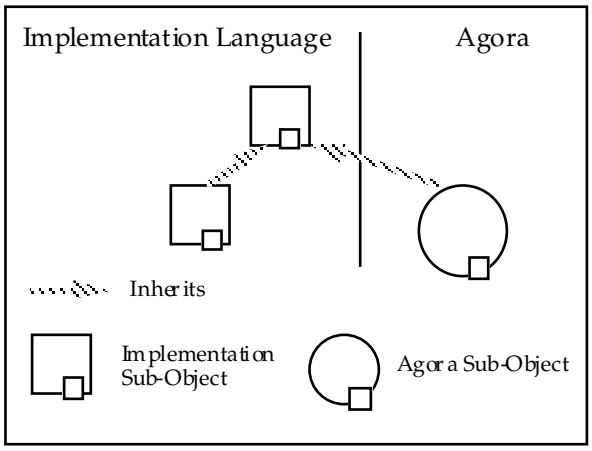

Figure 5.8

Since Agora's inheritance mechanism is prototype-, and mixin-based, technically, this can become non-trivial. We will not go into the technical details, but just give some indications of the problems involved.

Similarly to the symbiosis on the level of objects we need to identify how sub-objects are represented in the implementation. As we saw in chapter 4, sub-objects are represented internally by instances of concrete subclasses of 'AbstractInternalObject' that communicate with 'delegate:client:' messages. We can adopt the terminology of implicitly and explicitly encoded and referable sub-objects. We can also adopt a referent and representation relation on the level of sub-objects. This is shown in the next figure.

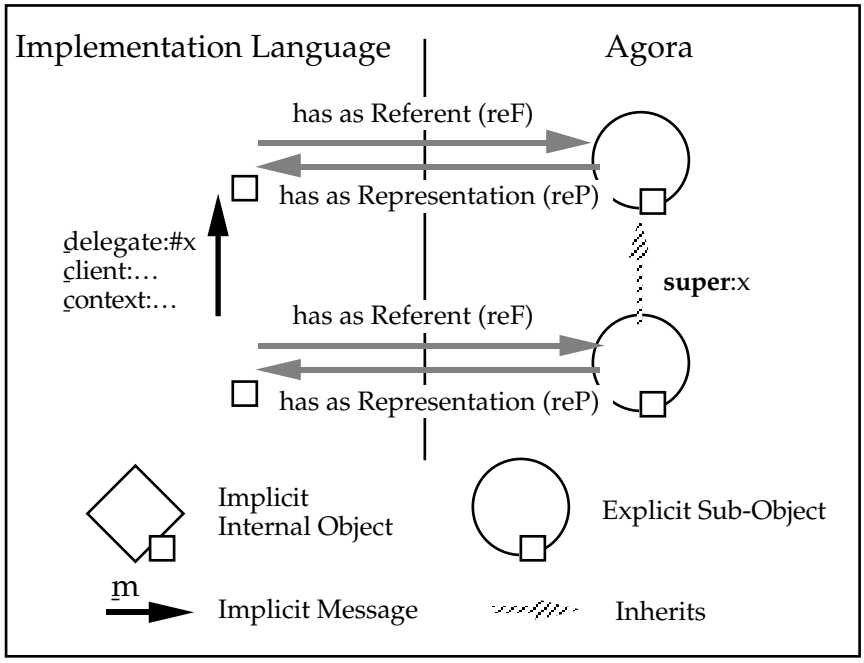

Figure 5.9

The problem now is that if we want explicitly encoded objects to inherit from implicitly encoded objects, then we need implicitly encoded objects that can be delegated to. Consider the following ideal situation. Presume that all implicitly encoded objects (e.g. 'AbstractExpression', 'Pattern', 'AbstractMetaObject', …) are such that messages can be delegated to them. In that case a variant of the 'asExplicit' conversion method can be devised such that implicit objects can be made explicit as sub-objects of explicit objects. This is depicted in figure 5.10.

So, in principle if implicitly encoded objects are objects that accept delegated messages then a symbiosis on the level of sub-objects between Agora and its implementations language can be realised. In practice however it is not possible to delegate messages to implicitly encoded objects. For this to be possible either the implementation language must be a language that supports message delegation or all objects that are part of the implementation must be encoded such that all methods have an extra set of delegation arguments (such as the 'self'). The former kind of languages were discarded in our analysis of object-orientation. The latter is an ad hoc implementation. An elegant solution to this problem remains an open issue.

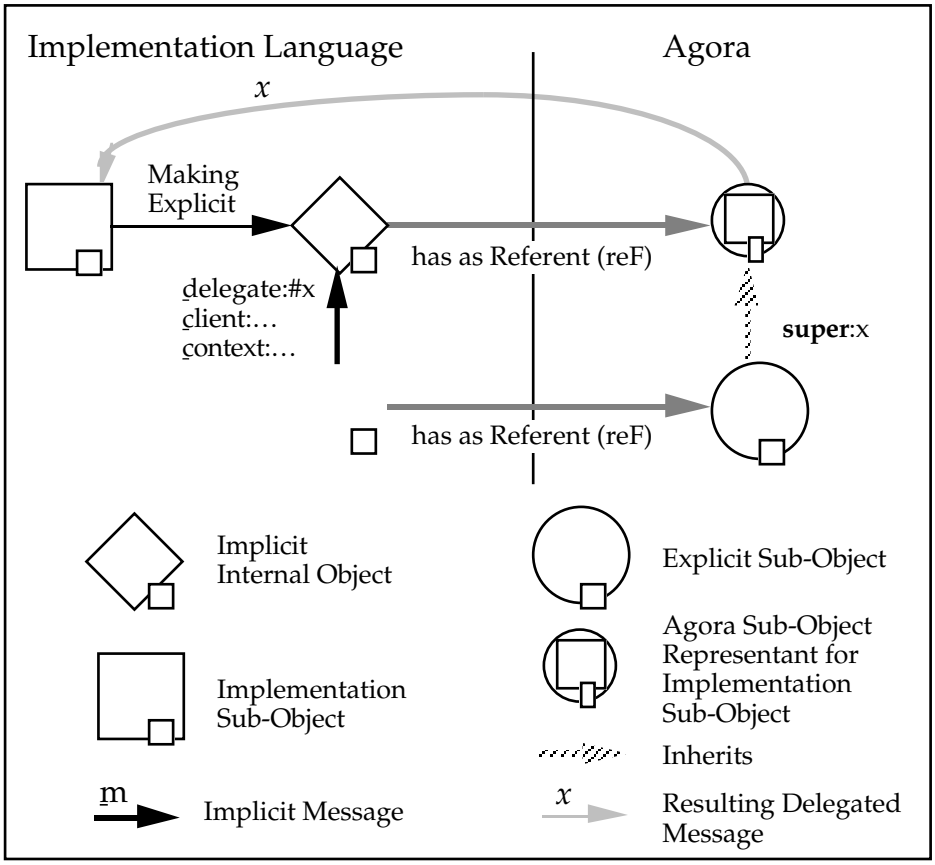

Figure 5.10

5.4 Conclusion and Open Issues

We conclude this section with some open issues regarding the introduction of reflection operators in an open implemented programming language.

Both of the above approaches to reflection in Agora (i.e. receiverless and other reifier messages) have their merits and their drawbacks. The second approach is too static but has the nice property of seamlessly integrating reflection and meta-programming. In the first approach reflection can be made more dynamic, but no explicit provisions are made for meta-programming.

Another apparent distinction is the management of extensions. The first approach is directed towards combining program pieces that are evaluated under different versions of the base language. The second approach may have better capabilities to combine different versions, but in its straightforward usage, programs, as a whole, must be expressed in the same extended language. In this situation differences may be trivially resolved, in the general case management of extensions of the base language may become an issue [Simmons&Friedman92] [Simmons&Friedman93].

We saw that for a considerably complex open implementation the choice of

reflection operators can become less than trivial. The design choices one has to make largely depend on the dynamic nature of reflection, and on how extensions of the base language are managed. One question that can be asked is whether reflection operators themselves can be used to introduce new reflection operators. It is trivial to see that all possible reflective facilities can be reconstructed in a reflective architecture (an example of a quote reifier is given below).

```
(AbstractExpression quote)
    using:context
    reifier: [ self return ]
```

More interestingly, in the case of Agora for example, the reifier declaration reifier itself can be reconstructed (meta-circularly):

```
(AbstractReifierPattern using:contextPattern reifier:body)
    using:declarationContext
    reifier:
      [declarationContext publicSlots add:
              (ReifierSlot  key:self
                            value:body
                            using:contextPattern) ]
```

It remains open to what extent and what useful reflective operators can be introduced reflectively.