OBJECT CLASSES, META CLASSES AND METHOD RESOLUTION IN OBJECT REXX

Rony G. Flatscher

Department of Management and Information Systems

Vienna University of Economics and Business Administration

„7th International Rexx Symposium", Austin/Texas, May 13th-15th, 1996

ABSTRACT

Object Rexx allows for defining Classes in a wide variety of ways. There are single and multiple inheritance available as well as the possibility to define one's own metaclasses to be used in the process of defining class hierarchies.

Definition of classes is possible via hardcoded „directives" or at runtime, fully dynamically! In addition one can define methods, which are not attached to a specific class, either hardcoded via directives or dynamically at runtime. These „floating" methods can be attached to individual objects and if so, define their own object scope.

Classes and methods offer interesting properties which are worthwile to get explored, including the ability of Object Rexx to define classes capable of interacting with SOM- or DSOM-objects.

1 CLASSIFICATION TREE, INSTANTIATION AND INHERITANCE

Derived from classification trees in biology software technology adopted the idea of organizing „chunks of code“ in form of a tree. Code and the data to go with it get encapsulated and are called classes, which get inserted at the appropriate place in the classification tree.

In a true object oriented environment every class inherits all parent classes up to the root, which is usually called the class „`Object`“. The class „`Object`“, being the root of the classification tree, of course has no superclass. The Object Rexx classification tree is shown in figure 1.

```
Object ........................................... The Class class
    Alarm ........................................ The Class class
    Array ........................................ The Class class
    Class ........................................ The Class class
    Directory .................................... The Class class
    DSOMServerProxy .............................. The Class class
    Envelope ..................................... The Class class
    List ......................................... The Class class
    Message ...................................... The Class class
    Method ....................................... The Class class
    Monitor ...................................... The Class class
    M_SOMProxy ................................... The M_SOMProxy class
    Queue ........................................ The Class class
    Relation ..................................... The Class class
        Bag ...................................... The Class class
    RX_QUEUE ..................................... The Class class
    sender ....................................... The Class class
    server ....................................... The Class class
    SOMProxy ..................................... The M_SOMProxy class
    Stem ......................................... The Class class
    Stream ....................................... The Class class
    Stream_Supplier .............................. The Class class
    String ....................................... The Class class
    Supplier ..................................... The Class class
    Table ........................................ The Class class
        Set ...................................... The Class class
```

Figure 1: Object Rexx' classification tree

A class may be *instantiated* which means that an individual „object“ is created which uses the „infrastructure“ laid out in the class (code and data), i.e. it is able to store data in „object variables“ and use the functions (called „methods“) defined in the class for manipulating the object variables. Each object is uniquely identified and insulated from all other objects, so it becomes even possible to instantiate several, individual objects from the *same* class.

Methods are invoked by sending objects messages which carry the name of the desired methods. The object invokes the appropriate method defined with the class it belongs to. If a method cannot be found within the class the object was instantiated from, then the search for an appropriate method proceeds sequentially thru those classes which are on the route to the root (these classes are called „superclasses"). The first superclass to contain the searched method is then used to carry out the message. Methods defined in superclasses are therefore said to be „inherited" by all subclasses.

If the classes are laid out „well" an extraordinary amount of reusage can be achieved by grouping tightly coupled functionality into separate classes. If the need of enhancing the functionality of a class arises this may be done by subclassing it and adding new methods and/or intercepting methods which are implemented in the superclass. Methods of a subclass may invoke any method in its superclass(es) by sending the appropriate message, possibly giving a class-selector to determine which class should be used as the starting point for the method resolution.

Subclasses are sometimes referred to as „specialisations" of their own superclass. By the same token a superclass may be called as a „generalisation" of all of its subclasses.

Figure 2 demonstrates a little Object Rexx program which defines the class „`Test`" with a single method named „`HI`" which in turn gets activated if an instance (object) of this class receives the message „`HI`". The class „`Test`" is a subclass of the root class „`Object`" and therefore inherits all of its methods. One of the methods defined with class „`Object`" is called „`DEFAULTNAME`" and returns a human readable string describing the type of the object. Because of this an instance of class „`Test`" understands the message „`DEFAULTNAME`" sent to it by forwarding it to its superclass, where it is defined. Figure 3 shows the output.

```
/* test.cmd */

anObject = .Test ~ new  /* create an instance (object)  */

/* send it messages & display result    */
SAY anObject ~ hi               /* send it "HI"         */
SAY anObject ~ defaultname      /* send it "DEFAULTNAME"*/

/* ---------------- class with methods --------------- */
:: CLASS Test   SUBCLASS Object

:: METHOD hi
   RETURN "hiii there, here we are !"

```

Figure 2: Defining a class, use inherited method from superclass "Object" too

```
hiii there, here we are !
a TEST
```

Figure 3: Defining a class, use inherited method from superclass "Object" too (output)

Methods defined for instances of a class are called „instance methods“, their object variables are called „instance object variables“.

2 METACLASSES AND INHERITANCE

So far we have been able to define a sort of blueprint for the object variables and methods for instances of any class in the Object Rexx classification tree.

2.1 Class Objects

As in Smalltalk Object Rexx adds the ability to interact with the *classes* themselves, by making them accessible as *plain* objects called „class objects“. For every class-definition Object Rexx will create exactly *one* instance of the class „`Class`“ by default, hence the name „class objects“. In addition it is possible to define „class methods“ and „class object variables“ for every class object.

Thruout this paper and the Object Rexx documentation the term „object“ is used for instances of any class. The term „class object“ is used for instances of the Object Rexx class „Class" only, which get automatically created by the runtime system as noted above.

Any object may get access to its „class object" by sending itself the message „CLASS" which is defined with the Object Rexx class „Object". This way *all* objects instantiated from a certain class will get access to that very same class object! Another way to get access to a specific class object is via an environment symbol which is created by merely prepending the class name with a dot. E.g. „.array" will return the class object for the class „Array".

Once a reference is established to a class object any messages sent to it get resolved in the usual way. That is to say, if a class object does not contain a method for a message sent to it, the class objects of its superclasses will get searched for up to the root of the classifcation tree, i.e. the object class for the class „Object". The interesting point here is that the class object of class „Object" is an instance of the Object Rexx class „Class" itself, hence method resolution will *continue* into the class methods of the class „Class"!

With other words there exists a *second* hierarchy tree specifically for the resolution of *class methods* (methods of class objects) ending in the Object Rexx class „Class". (Because „Class" is a subclass of „Object" one more method resolution will be undertaken, ultimately ending in the class object for „Object"; so one may conclude a circular relationship between the class objects for „Object" and „Class".)

The program in figure 4 exploits the concept of class object variables and class methods. The class object variable „myBabies" is an instance of class „Set" used to store all instances of class „someTest" within the class object. The class method „addToSet" adds the object it gets to „myBabies", the attribute class method „myBabies" allows for setting and retrieving the class variable object itself. The instance method „INIT" gets called whenever a „NEW" message is sent for creating an object. The „someTest" instance „INIT" method will establish access to its class object (via sending itself the „CLASS" message) and send the class object the message „addToSet" using itself as the argument. Finally in the initialisation part of that program three instances of „`someTest`“ are created by using cascading messages (with two twiddles) and then retrieving the class object variable „`myBabies`“ and sending it the „`Set`“-message „`ITEMS`“ to return the number of items presently in the set.

```
/* SomeTest.CMD */

.SomeTest ~~ new ~~ new ~~ new  /* create 3 instances   */
SAY .SomeTest ~ myBabies ~ items        /* shows "3"    */


/* ----------------- class with methods --------------- */
:: CLASS SomeTest

/* -------- class methods -------- */
:: METHOD init CLASS    /* class method                 */
   EXPOSE myBabies      /* class object variable        */

   myBabies = .set ~ new        /* create a "Set"-object*/

:: METHOD addToSet CLASS        /* class method         */
   EXPOSE myBabies
   USE ARG aBaby                /* get instance         */

   myBabies ~ put( aBaby )      /* put instance into set*/

:: METHOD myBabies CLASS ATTRIBUTE      /* class method */

/* -------- instance method ------ */
:: METHOD init                  /* called via NEW       */
   myClassObject = self ~ class /* get class object     */
   myClassObject ~ addToSet( self )     /* save object  */

```

Figure 4: Using class methods and class object variable

2.2 Metaclasses

The class „`Class`“ is special in another way too: its instance methods are at the *same time* its own class methods! This perceived circularity is necessary in order to close in on the class object for the class „`Class`“ itself.

As class objects are instances of the class „`Class`“ all instance methods of class „`Class`“ are available via inheritance. Therefore *all* classes have the instance methods of class „`Class`“ available as „class methods“. Of course this is true for the class „`Class`“ itself.

A class whose instance methods are its own class methods is called a metaclass. Therefore the Object Rexx class „`Class`“ *is* a metaclass. Any subclass of a metaclass is a metaclass itself. Instances of metaclasses may be called generically „metaclass objects“ (so the term „class objects“ and „metaclass objects“ could be intermixed) . By contrast, non-metaclasses are called „object classes“.

In Object Rexx there is a *second* metaclass defined (besides the class „`Class`“): the „`M_SOMProxy`“ class. This metaclass is used for creating objects representing (D)SOM-objects from the predefined class „`SOMProxy`“, thereby allowing for sending messages to (D)SOM-objects. Because Object Rexx' class „`Class`“ is part of the picture via the classification tree's root „`Object`“ it becomes possible to subclass (D)SOM-classes in terms of Object Rexx and use them as Rexx classes from then on.

Additionally, by subclassing class „`Class`“ it becomes possible to define one's own metaclasses. Added class methods and class object variables are only available from within the metaclass object of that metaclass. On the other hand the added instance methods become part of the class methods of all classes which use this newly created metaclass for their own creation. This may be done by using the „`METACLASS`“ attribute in the `::CLASS`-directive for those classes. In case a class omits the „`METACLASS`“ attribute, by default the metaclass of its parent class is used.

The rightmost column in figure 1 displays the metaclasses for the classes of the Object Rexx classification tree.

2.3 Multiple Inheritance

In contrast to Smalltalk Object Rexx allows for defining classes which have *more* than *one* class as their superclass. In object oriented terms this feature is called „multiple inheritance“. This way it becomes possible to *combine* different classes to form the basis for a new class.

Methods are resolved by searching first the superclass and its ancestors and if this fails (the method was not found), the classes (and their ancestors) given on the „INHERIT“-attribute are used as starting points for extending the search in the given order. Sending the „SUPERCLASSES“-method of the class „Class“ to the class object will show this order for all immediate superclasses.

Figure 5 shows three classes, named „RelTablelike“, „RelBijective“ and „RelDirlike“.

„RelTablelike“ is a specialized form of the Object Rexx supplied class „Relation“, which allows among other things to get an item for a given index, or to determine which index refers to a certain item (which was not possible with the Object Rexx „Table“ class). The only restriction defined in „RelTablelike“ deals with the method of putting a new item into the relation (there are two methods defined for this behavior in class „Relation“, namely „PUT“ and „[]=“), thereby assuring the table-property that there exists one item only with a specific index. To make sure that an index does not contain more than one item, by default the index is deleted, before the original „PUT“-method of class „Relation“ adds the item to the relation itself.

```
/* multi_inh.cmd */

aClassObject = .RelTablelike ~ subclass( "<Table & Directory Like>" )
SAY aClassObject                        /* show Defaultname     */

aClassObject ~ inherit( .RelDirlike )   /* additional inheritance       */
anObject = aClassObject ~ new           /* create instance      */
SAY anObject                            /* show Defaultname     */

anObject ~ Tyrol = "some country in Austria"    /* add an item  */
SAY anObject ~ Tyrol                            /* display entry*/


:: CLASS RelTablelike       SUBCLASS Relation       PUBLIC

:: METHOD "[]="                 /* override             */
   FORWARD MESSAGE ( "PUT" )    /* let PUT do the work  */

:: METHOD "PUT"                 /* override             */
   USE ARG item , index
   self ~ remove( index )       /* remove index & associated item   */
   FORWARD CLASS (super)        /* now do the PUT !     */

```

```
:: CLASS RelBijective   SUBCLASS Relation       PUBLIC

:: METHOD "[]="                     /* override             */
   FORWARD MESSAGE ( "PUT" )     /* let PUT do the work  */

:: METHOD  PUT                      /* override             */
   USE ARG item , index
   self ~ remove( index )        /* remove index         */
                  /* remove item (with another index): */
   self ~ removeitem( item , self ~ index( item  ) )
   FORWARD CLASS (super)         /* let super do the PUT !           */

/* .directory-like, because of enhancing with SETENTRY, ENTRY, HASENTRY */
:: CLASS RelDirlike          MIXINCLASS Relation     PUBLIC

:: METHOD  ENTRY        /* returns the item associated with index   */
   USE ARG index
   RETURN self ~ at( TRANSLATE( index ) )        /* use uppercase    */

:: METHOD  SETENTRY     /* "name" is used as index                  */
   USE ARG name, value
   self ~ PUT( value, TRANSLATE( name  ) )      /*  uppercase it    */

:: METHOD  UNKNOWN      /* define an unknown method                 */
   USE ARG messageName, messageArgs

   IF RIGHT( messageName, 1 ) = "=" THEN        /* setentry method  */
   DO                   /* remove trailing "=" from message name    */
      index = LEFT( messageName, LENGTH( messageName ) - 1 )
      FORWARD MESSAGE ( "SETENTRY" ) ARRAY ( index, messageArgs[ 1 ] )
   END
   ELSE                                         /* entry method */
      FORWARD MESSAGE ( "ENTRY" )    ARRAY ( messageName )
```

Figure 5: An example for multiple inheritance

„`RelBijective`“ is another variant of a relation which makes sure that an index has just one entry (making it tablelike) and that an item occurs just once in the entire collection.

„`RelDirlike`“ is a class which implements the three methods directories possess in order to allow for conveniently using strings as indices: „`ENTRY`“, „`SETENTRY`“ and „`UNKNOWN`“. The logic is simple: any message not understood (not found in the class or its ancestors) is intercepted via the „`UNKNOWN`“-method. The message name is uppercased and interpreted as a string index. If this message contains a trailing equal sign („`=`“) it is assumed that the programmer wants to add an item associated with an uppercased string (the name of the method), otherwise it is assumed that the programmers want to retrieve the item associated with the uppercased string (the name of the method). In the former case the „`SETENTRY`“-method is run; in the latter the „`ENTRY`“-method, exactly the way the Object Rexx class „`Directory`“ behaves.

The program creates a new class („`<Table & Directory Like>`“) at runtime by subclassing „`RelTablelike`“. This is done by sending the method „`SUBCLASS`“ to the class object. The next step is to add multiple inheritance to that freshly created class object by issuing an „`INHERIT`“ message denominating the additional superclass „`RelDirlike`“ as an argument. From this moment on methods are looked up not only in class „`RelTablelike`“, but in „`RelDirlike`“ too. This new class now behaves like a table *and* like a directory, yet all methods of the class „`Relation`“ are available too, notably the ability to query an index for an existing item. Figure 6 shows the output of our example in figure 5.

```
The <Table & Directory Like> class
a <Table & Directory Like>
some country in Austria
```

Figure 6: An example for multiple inheritance (output)

To create class „`<Table & Directory Like>`“ via a directive would look like:

```
::CLASS "<Table & Directory Like>" SUBCLASS RelTablelike INHERIT RelDirlike
```

Should the programmer wish to add the directorylike functionality of „`RelDirlike`“ to the „`RelBijective`“ class all that is necessary is sending the „`INHERIT`“ message or using the „`INHERIT`“ attribute in the class directive.

Note that the class directive for „`RelDirlike`“ uses „`MIXINCLASS`“ rather than „`SUBCLASS`“ to determine that its parent class is the class „`Relation`“. By using „`MIXINCLASS`“ the programmer indicates to the runtime system that this class may be used as an additional superclass for classes which descend from the same „baseclass“. The baseclass in turn is defined to be the first class in the superclass ancestors which was not created with the „`MIXINCLASS`“-attribute but with the „`SUBCLASS`“-attribute instead.

Therefore only classes descending from the Object Rexx Class „`Relation`“ may utilize the class „`RelDirlike`“ as an additional superclass for multiple inheritance.

3 CREATING CLASSES

There are two ways to create Object Rexx classes:

- statically via directive statements or
- dynamically via messages to class objects.

Figure 5 demonstrates both possibilities, creating classes via directives („`RelTablelike`“, „`RelBijective`“ and „`RelDirlike`“) and dynamically („`<Table & Directory Like>`“). In essence the methods defined with the Object Rexx class „`Class`“ are used for the purpose of creating classes on the fly.

As a matter of fact, once the runtime system of Object Rexx has been initialized, Object Rexx itself uses messages of the „`Class`“ class to turn e.g. class directives of parsed Object Rexx programs into full fledged class objects. While initialising Object Rexx creates an instance for the class „`Class`“ which gets used for creating all other Rexx supplied class objects, thereby creating the classification tree.

Classes are built from class directives right after the interpreter has successfully parsed an Object Rexx program source and *after* all the „`::REQUIRES`“-directives (if any) have been resolved (recursively if necessary).

The parent class objects of the new classes to be built are used for creating new class objects by sending the „`SUBCLASS`“ (or „`MIXINCLASS`“) message to the superclass class object.

During the creation process the new class object gets the (class) method „`INIT`“ sent. Note, that at this particular point multiple inheritance is *not* setup yet, hence class method resolutions within the class method „`INIT`“ will only go up to its immediate superclass. This is of interest *only*, if the programmer was to override the superclass' class method „INIT“ with an own class method. If he/she wishes to do so it is recommended to forward the class method „INIT“ up to the superclass so it is able to do its part of initialisation. Once the class was setup successfully (upon returning from the class method „INIT“) multiple inheritance - if defined for that class - is incorporated via „INHERIT“ messages by the runtime system and thereafter available for method resolution.

4 DEFINING AND ASSIGNING METHODS

As has been seen in the figures (e.g. figure 5) methods may be defined statically in the source of an Object Rexx program by means of the „::METHOD“ directive.

4.1 Defining Methods Dynamically

To define a method dynamically it is necessary to create an instance of the Object Rexx class „Method“ by sending the message „NEW“ to the method class object giving the name of the method and the Rexx source code as arguments. The initialisation process will parse the source code and if successful will return a method object.

4.2 Assigning Methods

If methods are defined statically via directives those preceeding the very first class directive are collected by Object Rexx and put into a directory accessible via the environment symbol „.methods“ by that Object Rexx program. Because such methods are not assigned to a class, they may be regarded as „floating“ methods.

```
/* def_meth.cmd */

Object1 = .Test ~ new

/* add floating method to object:                              */
Object1 ~ setmethod( "method1", .methods ~ entry( method1 ))
Object1 ~ method1                       /* separate object scope*/
SAY
/* add method to class object:                                 */
RexxCode = "EXPOSE variable                    ;" ,
           "IF \ VAR( 'variable' ) THEN        ;" ,
           "   variable = 'value from RexxCode' ;" ,
           "SAY 'RexxCode:' variable           ;" ,
           "SAY 'self:' self 'super:' super      "
```

```
.Test ~ define( "method2", RexxCode )    /* add method to class   */
Object2 = .Test ~ new                    /* new object gets it    */
Object2 ~ method2                        /* .Test object scope    */
SAY

/*  defining a floating method: */
:: METHOD method1
   EXPOSE variable

   IF \ VAR( 'variable' ) THEN
      variable = 'value from method1'
   SAY 'METHOD1:' variable
   SAY 'self:' self 'super:' super

:: CLASS Test

:: METHOD init
   EXPOSE variable
   variable = 'value from .Test'
   SAY 'class Test, method INIT, variable set to:' variable
   SAY 'self:' self 'super:' super
   SAY
:: METHOD setmethod              /* allow for using SETMETHOD */
   FORWARD CLASS ( super )
```

Figure 7: Assigning methods with SETMETHOD and DEFINE

All method directives following a class directive are assigned to the immediately preceeding class directive, thereby forming the methods a class will gain.

In addition there are many ways to add methods to a class object (cf. the methods „`DEFINE`“, „`MIXINCLASS`“ and „`SUBCLASS`“ of the Object Rexx class „`Class`“) and even to individual objects (cf. the method „`SETMETHOD`“ of class „`Object`“ and „`ENHANCED`“ of class „`Class`“).

Figure 7 shows a program which adds the floating method „`METHOD1`“ to the *object* „Object1“ only. Because producing one-off-objects (objects which possess methods which are not defined with the appropriate class object) might be dangerous in some situations the „`SETMETHOD`“ method of class „`Object`“ is defined to be private. A private method can only be invoked by sending it to itself, therefore a public „`SETMETHOD`“ method was defined for class „`.Test`“ which forwards this message to its own superclass, effectively resending itself the message but taking care that a superclass method (from class „`Object`“ in this case) is chosen.

```
class Test, method INIT, variable set to: value from .Test
self: a TEST super: The Object class

METHOD1: value from method1
self: a TEST super: The TEST class

class Test, method INIT, variable set to: value from .Test
self: a TEST super: The Object class

RexxCode: value from .Test
self: a TEST super: The Object class
```

Figure 8: Object scope of methods assigned with SETMETHOD and DEFINE (output)

5 OBJECT SCOPES

If looking at the output in figure 8 it becomes clear that the *object scope* (Flatscher, 1996) is *different* for method „`METHOD1`“ and method „`METHOD2`“. This can be seen by looking at the different value of the pseudo variable „`super`“ which points to the class object (towards the root of the class tree) in which method resolution proceeds by default. Another clue is watching the value of the object variable „variable“ which is different too, because the object variables reside in different object scopes .

Adding methods to *instances* (with `SETMETHOD` or `ENHANCED`) always causes these methods to execute within their own, *separated* object scope together with the object variables they define; the pseudo variable „`super`“ points to the class object of the instance rather than to the class object's superclass which would be the class object for class „`Object`“ in this particular example.

Adding methods to *class objects* (with `DEFINE`, `SUBCLASS` or `MIXINCLASS`) incorporates these methods and the object variables these methods may define into the object scope of the *class object itself*. Instances created thereafter will have these added methods available at the class' object scope! As an example see „`METHOD2`“ which gets defined at runtime for the class object of the class „`Test`“ and is available for object „`Object2`“ in figure 7 and 8, which got instantiated right after adding that new instance method. Looking at the value of the pseudo variable „`super`“ reveals that the next class object to look up messages would be the one representing the class „`Object`", because this is „`Test`"'s superclass.

6 INTERFACING WITH SOM

As Object Rexx is fully SOM enabled it is a simple task to utilize SOM or DSOM for that matter. One merely needs to define one Object Rexx class using the attribute „`EXTERNAL`" indicating that access to SOM is desired and which class to use. At this moment the SOM environment becomes accessible and may be used e.g. to query the SOM interface repository which contains all SOM classes, their methods, arguments and return values. It is easy to determine which part of the interface definitions one wants to inspect.

```
/* querying the SOM interface repository with Object REXX */
aRepository = .somClassMgrObject ~ _get_somInterfaceRepository

SAY "repository:" pp(aRepository) "of class:" pp(aRepository ~ class)
SAY

aContainer = aRepository ~ contents( "InterFaceDef", .true )
SAY "aContainer:" pp( aContainer ) "items" pp( aContainer ~ items )

i = 0
length = LENGTH( aContainer ~ items )
DO anItem OVER aContainer       /* loop over array */
   i = i + 1
   SAY RIGHT( i, length) "id:" LEFT( pp( anItem ~ _get_id), 35 ),
      "name:" pp( anItem ~ _get_name )
   anItem ~ somFree             /* free SOM object in hand        */
END

aRepository ~ somFree           /* free the repository SOM object */

:: ROUTINE pp; RETURN "[" || arg( 1 ) || "]"

/* --------------- class to get access to SOM ---------- */
::CLASS Test                          PUBLIC  EXTERNAL        'SOM SOMObject'
```

Figure 9: Accessing the SOM interface repository and querying it

Object Rexx uses and supplies the class definitions found in the program „`DLFClass.CMD`" for dealing with SOM datatypes. This file may be found in Object Rexx' home directory.

The program in figure 9 gets access to the SOM environment by defining the class „`Test`" for representing the SOM class object „`SOMObject`". Classes which represent other classes are called „proxies". A proxy object forwards messages it does not understand to the object it represents. Figure 10 shows part of the possible results of querying the installed SOM-repository.

```
repository: [a Repository] of class: [The SOMProxy class]

aContainer: [an Array] items [374]
  1 id: [::SOMObject]                          name: [SOMObject]
  2 id: [::Sockets]                            name: [Sockets]
  3 id: [::AnyNetSockets]                      name: [AnyNetSockets]
  4 id: [::Contained]                          name: [Contained]
  5 id: [::AttributeDef]                       name: [AttributeDef]
  6 id: [::BOA]                                name: [BOA]
  7 id: [::SOMEEvent]                          name: [SOMEEvent]
  8 id: [::SOMEClientEvent]                    name: [SOMEClientEvent]
  9 id: [::Context]                            name: [Context]
 10 id: [::ConstantDef]                        name: [ConstantDef]
 11 id: [::Container]                          name: [Container]
 12 id: [::SOMPEncoderDecoderAbstract]         name: [SOMPEncoderDecoderAbstract]
 13 id: [::SOMPAttrEncoderDecoder]             name: [SOMPAttrEncoderDecoder]
 14 id: [::SOMEEMan]                           name: [SOMEEMan]
 15 id: [::SOMEEMRegisterData]                 name: [SOMEEMRegisterData]
 16 id: [::ExceptionDef]                       name: [ExceptionDef]
 17 id: [::SOMPMediaInterfaceAbstract]         name: [SOMPMediaInterfaceAbstract]
 18 id: [::SOMPFileMediaAbstract]              name: [SOMPFileMediaAbstract]
 19 id: [::SOMPAsciiMediaInterface]            name: [SOMPAsciiMediaInterface]
 20 id: [::SOMPBinaryFileMedia]                name: [SOMPBinaryFileMedia]
 21 id: [::SOMPIOGroupMgrAbstract]             name: [SOMPIOGroupMgrAbstract]
 22 id: [::SOMPAscii]                          name: [SOMPAscii]
 23 id: [::SOMPBinary]                         name: [SOMPBinary]
 24 id: [::ImplementationDef]                  name: [ImplementationDef]
 25 id: [::ImplRepository]                     name: [ImplRepository]
 26 id: [::InterfaceDef]                       name: [InterfaceDef]
 27 id: [::SOMPKeyedSet]                       name: [SOMPKeyedSet]
 28 id: [::SOMPIOGroup]                        name: [SOMPIOGroup]
 29 id: [::IPXSockets]                         name: [IPXSockets]
 30 id: [::dictKeyCharPjw]                     name: [dictKeyCharPjw]
 31 id: [::SOMRLinearizable]                   name: [SOMRLinearizable]
 32 id: [::somf_MCollectible]                  name: [somf_MCollectible]
 33 id: [::somf_MLinkable]                     name: [somf_MLinkable]
 ------------------------------ cut ------------------------------------
```

Figure 10: Accessing the SOM interface repository and querying it (output)

7 INTERFACING WITH THE WORKPLACE SHELL

Because OS/2's Workplace Shell is a user interface framework built with SOM, it becomes possible for Object Rexx programs to interact with WPS-objects by sending them the appropriate *WPS*-messages. As with SOM classes it is possible to specialise the behavior of WPS-classes by subclassing them and adding/overriding the desired methods. This way *other* WPS objects and WPS aware programs may start to utilize WPS-objects which in fact were created by Object Rexx and have Rexx code running in them! The Object Rexx developer even supplied a DLL to make interaction with the WPS even simpler, by allowing direct reference via Object Rexx without the need to define the „`EXTERNAL`" attribute on class directives.

Figure 11 shows an example Rick McGuire once posted in the internet newsgroup „`comp.lang.rexx`" which implements a password protected WPS-folder in *Object Rexx*! This is done by specialising the WPS-folder object class with features (behaviors) described in terms of Object Rexx.

The logic is fairly simple:

- class „`VXPWPrompts`" handles input/output thru an OS/2 window (used later on for setting the password) with the help of VX-Rexx in this example,
- class „`WPLockFolder`" subclasses the WPS-folder class object and merely „catches" those WPS-messages (starting with the string „`wp`") directed to the WPS-folder which it is interested in; the two most important messages are of course the creation of a password protected folder („`INIT`") and opening („`wpOpen`") of such folders; note that „`WPLockFolder`" uses multiple inheritance to gain access to the functionality (behavior) the „`VXPWPrompts`" class defines.
- class „`SMPPWChange`" is used to place an icon in the password protected folder, which - if opened (double-clicked) by the user - causes the password change dialog (associated via multiple inheritance with the „`WPLockFolder`" object) to be displayed by sending the appropriate message to the password protected folder object.

```
/* */
call RXFuncAdd "VRLoadFuncs", "VROBJ", "VRLoadFuncs"
call VRLoadFuncs
environment['WPLOCKFOLDER'] = .WPLockFolder

::CLASS VXPWPrompts mixinclass object
::METHOD init
 expose buttons. default esc
  self~init:super                       /* let super class initialize.       */
  self~initButtons

::METHOD initButtons
 expose buttons. default esc
  default = 1                           /* Set up some default values for    */
  esc = 2                               /* the VX-REXX windows....           */
  buttons.0 = 2
  buttons.default = 'Ok'
  buttons.esc     = 'Cancel'

::METHOD displayPrompt
 expose buttons. default esc
 use arg prompt, title
                                        /* Display Password prompt           */
  rc = VRPrompt('', prompt, 'ENTERPW', title, 'BUTTONS.', default, esc)
  if rc = esc then                      /* cancel requested?                 */
   return .nil                          /* Yup, indicate no PW entered       */
  else
   return enterPw                       /* Return entered PW.                */

::METHOD displayError
 expose buttons. default esc
 use arg title
                                        /* Display error message for         */
                                        /*incorrect PW                       */
  return VRMessage('', 'Incorrect Password Entered !!', title, 'E', 'BUTTONS.', default, esc)

/* ********************************************************* */
::CLASS WPLockFolder subclass WPFolder INHERIT VXPWPrompts
::METHOD wpclsQueryTitle CLASS
  return 'LockFolder'

::METHOD init
 expose password
  self~init:super                       /* let superclass initialize 1t      */
                                        /* Create object to allow PW Change  */
  .smpPwChange~new('Change Password', 'ICONFILE=C:\OBJREXX\PMREXX.ICO', self, 1)
  if \var('PASSWORD') Then            /* PW initialized via SetupString?   */
    password = ''                     /* Nope, give default ''             */
```

```
::METHOD wpOpen
 expose password
 use arg handleContainer, view, params
 if password == '' then                /* no password set?                 */
   return self~wpOpen:super(handleContainer, view, params) /* go ahead and open this*/
                                       /* Ask user for password.           */
  enterpw = self~displayPrompt('Enter Password ...','Locked Folder Password')
  if password = enterPw then Do        /* Was correct password entered     */
                                       /*Yup, forward to WPFolder top Open */
   return self~wpOpen:super(handleContainer, view, params)
  End
  else Do                              /* Incorrect pw entered.            */
   reply .false                        /* Return failure, and return to WPS */
   guard off
   self~displayError('LockFolder Error' password) /* Now display error to user. */
  End

::METHOD wpSetup
 use arg setupString
 strLength = .WPDLFULong~new(256)      /* Will allow for up to 256 cahr PW  */
                                       /* Get INOUT String parm            */
 str = .WPDLFString~new~~_set_maxSize(strLength)
                                       /* see if setup strings has PW      */
 if self~wpScanSetupString(setupString, 'PASSWORD', str, strLength)  then
   self~password = str~asString        /* Yup, set password.               */

 return self~wpSetup:super(setupString)/* Superclass does remainder.       */

::METHOD scrollTitle unguarded         /* unguarded, want to rn concurrently*/
 title = self~wpQueryTitle             /* Get current title                */
 do 2                                  /* Will scroll twice.               */
   do i = 1 to title~length            /* For length of title.             */
                                       /* display 1st 1 chars of title     */
      self~wpSetTitle(right(left(title, i), title~length))
   end                                 /* iincrement i.                    */
 end

::METHOD password ATTRIBUTE            /* set/retrieve methods for password */

::METHOD wpSaveState                   /* Save the password data           */
 self~wpSaveString(self~somGetClassName, 1, self~password)
 return self~wpSaveState:super         /* Let parent save any state.       */

::METHOD wpRestoreState
 self~initButtons                      /* make sure OREXX side initialized. */
 size = .WPDLFULong~new                /* Get DLFUlong for size query.     */
                                       /* Retrive size of string for restor */
 self~wpRestoreString(self~somGetClassName, 1, .nil, size)
                                       /* Create DLFString large enough to  */
                                       /*  contain the string, plus NULL    */
 str = .WPDLFString~new~~_set_maxSize(size+1)
                                       /* Now get saved password.          */
 self~wpRestoreString(self~somGetClassName, 1, str, size)
 self~password = str~asString          /* Save password state value.       */
                                       /* let parent restore state.        */
 return self~wpRestoreState:super(arg(1))
```

```
::CLASS SMPPWChange subclass WPAbstract
::METHOD wpOpen
 use arg handleContainer, view, params
 if view \= 2  & view \= 3 then Do      /* Opening Default view? Dbl-click   */
   lockf = self~wpQueryFolder           /* Get our conatining lock folder    */
                                        /* Ask for new password              */
   newpw = lockf~displayPrompt('Enter New Password', 'New LockFolder Password')
   if newpw \= .nil Then Do             /* Get a new password?               */
     lockf~password = newpw             /* Yup, set new pw.                  */
     lockF~wpSaveImmediate              /* Save object state (PW)            */
   End
   return 0
 End
 else                                   /* Forward wpOpen to super class to handle. */
   return self~wpOpen:super(handleContainer, view, params)
```

Figure 11: Password Protected Folder in Object Rexx (source: Rick McGuire, posted in the internet newsgroup on "Date: 10 Aug 1995 22:18:09 GMT", "Subject: Re: A practical example of ORexx?")

All WPS-messages not intercepted by the Object Rexx program are handled by the appropriate WPS classes due to the OO-method resolution. This example shows how powerful true object orientation (in this case inheritance) is, even more, how impressive the object oriented features of Object Rexx are! It should also demonstrate how Object Rexx extends the notion of scripting into the object oriented world by allowing it to be used as a scripting language for object oriented systems.

8 TWO LITTLE UTILITIES RELATED TO CLASSES: ISA AND ISCLASS

While developing Object Rexx programs many times it becomes necessary to be able to tell whether an object is of a specific class (or its superclasses), and at times it may become important to learn at runtime whether an object in hand is a *class object*.

Figure 12 depicts two routines where „`IsA`“ determines whether an object (or a class object) belongs to the class of the object given as the second argument. The routine „`IsClass`“ determines whether an object is in effect a class object by attempting to send it messages which allow for determining this fact.

There are also two floating methods defined, which may be used to be defined dynamically to classes and which allow for using the message form „`self ~ IsA( otherClass )`“ and „`self ~ IsClass`“.

```
/* IsA_IsClass.cmd */

:: METHOD IsA                          /* see routine for description */
   USE ARG SuperClassID
   RETURN isA(self, SuperClassID) /* call routine to do the work */

:: METHOD IsClass                      /* see routine for description */
   RETURN isClass(self)            /* call routine to do the work */

:: ROUTINE IsA                       PUBLIC
   USE ARG SubClass, SuperClassID, internal_flag

   /* execute on very first invocation only ("internal_flag" is undefined),
      make sure to turn an object argument into its own class object for
      testing     */

   IF internal_flag <> .true THEN       /* check for class objects            */
   DO
       IF \IsClass(SubClass)      THEN SubClass     = SubClass ~ class

       IF \IsClass(SuperClassID) THEN SuperClassID = SuperClassID ~ class ~ ID
                                 ELSE SuperClassID = SuperClassID ~         ID
   END

   IF SubClass~ID = SuperClassID THEN   /* o.k., same class                   */
      RETURN .true

   DO SuperClass OVER SubClass~SUPERCLASSES    /* test immediate superclasses */
      IF SuperClass~ID = SuperClassID THEN
         RETURN .true
   END

   /* not found, maybe one of the superclasses preceding the immediate ones ? */
   DO SuperClass OVER SubClass~SUPERCLASSES
      IF IsA(SuperClass, SuperClassID, .true) THEN
         RETURN .true
   END

   RETURN .false                        /* nope                               */
```

```
:: ROUTINE IsClass                PUBLIC
   USE ARG object

   /* if argument is a plain object NOMETHOD will be raised  */
   SIGNAL ON  NOMETHOD NAME nomethod_IsClass
   tmp =  object ~ QUERYMIXINCLASS      /* try class' method QUERYMIXINCLASS  */

   /* take care of the case that we have an object of type .directory in hand
      which has an interesting UNKNOWN method implemented                    */
   SIGNAL ON  NOMETHOD NAME nomethod_IsClass2
   object ~ ENTRY( "id" )               /* use a .directory instance method   */

   RETURN .false        /* testing a directorylike object !                  */

NOMETHOD_IsClass:       /* not a class object, method QUERYMIXINCLASS unknown */
   RETURN .false

NOMETHOD_IsClass2:      /* a class object, does not understand the directory message*/
   RETURN .true
```

Figure 12: Determining whether (class) object is of a specific kind and/or if object in hand is a class object

9 CONCLUSIONS

This paper attempted to discuss various aspects of classes, metaclasses and method resolutions as implemented by Object Rexx. Due to a true object oriented implementation of the language all the benefits object oriented systems may be capable of taking advantage of are available to programmers of Object Rexx. Because the language has been kept as simple as possible, yet became extremely powerful in terms of object orientation it has the potential of revolutionizing scripting languages with respect to object oriented systems.

It is therefore a logical choice for IBM that Object Rexx is being used as the preferred scripting language for OpenDoc's „open scripting architecture“ (abbrev. OSA) in its own OpenDoc implementations.

If the language gets ported to the various Unix flavors (most notable Linux and AIX) enhanced with a standard class for utilizing (TCP/IP) sockets it may even be a stark contender with Unix scripting languages like Perl, Python or TcL, allowing Object Rexx to become an industry wide scripting standard. As a matter of fact in this case even a *network* scripting language.

Object Rexx in any case allows for utilizing all CORBA services via the SOM interface, adding additional potential to the language. The moment Java classes get SOMified (or „CORBAfied“) it became possible to utilize Object Rexx as a scripting language for Java applets and applications also.

Given such potential developments Object Rexx would be able to become a true universal scripting language in the IT-market.

10 ACKNOWLEDGEMENTS

The author wishes to thank Rick McGuire, one of the former lead developers for Object Rexx, for his great hints and explanations given on the Internet-newsgroup „comp.lang.rexx“ thru the past years, and Christian Michel the present lead developer for the Object Rexx project.

11 REFERENCES

Online documentations of various beta versions of Object Rexx (the latest version used for this paper stems from February 16th, 1996).

Various postings on the internet newsgroup „comp.lang.rexx“, 1995-1996.

Flatscher, R.G.: „Local Environment and Scopes in Object Rexx“, in: Proceedings of the "7th International REXX Symposium, May 12-15, Texas/Austin 1996", The Rexx Language Association, Raleigh N.C. 1996.

Goldberg, A., Robson, D.: „Smalltalk-80 - The Language and its Implementation“, Reading: Addison Wesley, 1983..

Date of Article: 1996-05-30.

Published in: Proceedings of the "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association, Raleigh N.C. 1996.

Presented at: "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association.