Scripting en Rational Rose
(Versión borrador)

Octavio Martín Díaz
Dpto. de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadística, Universidad de Sevilla
Avda. de la Reina Mercedes s/n, Sevilla 41.012
Tel/Fax: 95.455.71.39, e-mail: octavio@lsi.us.es

26 de Enero de 2.000

Capítulo 1

Introducción

El presente manual se ha realizado a partir de la ayuda ofrecida por RATIONAL ROSE 98.

1.1 Extensibilidad de RATIONAL ROSE

RATIONAL ROSE permite la posibilidad de extender y personalizar su funcionalidad a aquellas necesidades de desarrollo de software específicas para un proyecto determinado. Son entre otras:

- Automatizar las acciones del usuario en el entorno de RATIONAL ROSE con *scripts* (por ejemplo, creación de diagramas y clases, modificaciones del modelo, generación de documentos y otras).
- Ejecutar las funciones de ROSE desde otras aplicaciones mediante la automatización OLE.
- Acceder a las clases, propiedades y métodos de un modelo de ROSE desde un entorno de desarrollo de software mediante la inclusión de la librería de extensión de tipos.
- Personalizar los menús de la aplicación.
- Utilizar el administrador de anexos (*Add–ins*).

1.1.1 *Scripting*

El lenguaje de *scripting* de RATIONAL ROSE es una versión ampliada del lenguaje SUMMIT BASICSCRIPT. Las extensiones realizadas a este lenguaje permiten automatizar las acciones del usuario en el entorno de la aplicación y en algunos casos realizar funciones que no están disponibles directamente a través del interfaz de usuario.

1.1.2 Automatización OLE

La automatización OLE permite la integración de otras aplicaciones con RATIONAL ROSE de dos formas:

- Utilizando ROSE como un controlador de la automatización: un objeto OLE puede ser llamado desde un *script*. Por ejemplo: un *script* puede ejecutar funciones en aplicaciones tales como WORD o VISUAL BASIC.

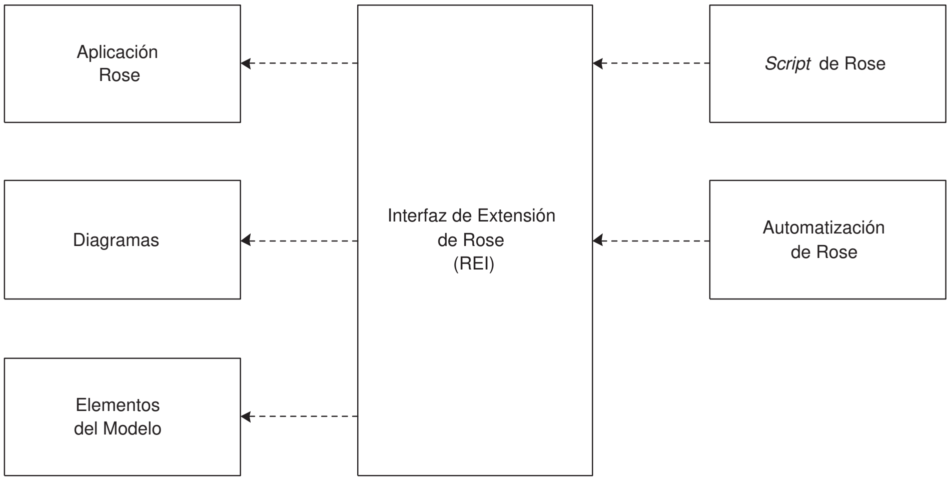

Figura 1.1: El diagrama de componentes de Rational Rose.

- Utilizando Rose como un servidor de automatización: el objeto OLE de la aplicación puede ser llamado desde otras aplicaciones que actúen como un controlador OLE. Por ejemplo: Visual Basic, Excel, Summit BasicScript, Softbridge Basic Language, C++ y otros.

1.1.3 El modelo de componentes de Rational Rose

La aplicación Rational Rose está basada en componentes y está enfocada hacia el desarrollo de software basado en componentes. Toda la funcionalidad de la aplicación es accesible a través del Modelo del Interfaz de Extensión de Rose (REI) que esencialmente es un metamodelo de un modelo de Rose que expone los paquetes, clases, propiedades y métodos que definen y controlan la aplicación y todas sus funciones.

La comunicación con este interfaz se realiza a través de *scripts* o de la automatización OLE. La librería de extensión de tipos *RationalRose.tlb* permite acceder al interfaz REI desde un entorno de programación.

El diagrama de la figura 1.1 muestra los componentes del núcleo de Rational Rose, los componentes del interfaz REI y las relaciones entre ellos:

- **Interfaz de extensión** El conjunto común de interfaces utilizados por un *script* o una automatización OLE para acceder a Rational Rose. A su vez se subdividen en tres interfaces muy importantes:
  - *Aplicación Rose* Los objetos de extensión que forman el interfaz a las funciones de la aplicación.
  - *Diagramas* Los objetos de extensión que forman el interfaz a los diagramas y vistas.
  - *Elementos del modelo* Los objetos de extensión que forman el interfaz a los elementos de un modelo.
- ***Scripting*** El conjunto de *scripts* que incrementan la funcionalidad de Rational Rose.
- **Automatización** El conjunto de objetos de automatización que permiten a Rational Rose funcionar como un controlador o un servidor OLE.

1.1.4 Personalización de los menús

los menús de la aplicación se pueden extender o adaptar modificando el fichero de menús (*Rose.mnu*) que ROSE lee al inicio de su ejecución. Se permiten añadir:

- Submenús.
- Opciones de menú que ejecuten:
  - Primitivas de ROSE.
  - *Scripts*.
  - Comandos de sistema.
  - Programas externos.
- Separadores de menús.

1.1.5 Administrador de anexos (*add–ins*)

El administrador de anexos de RATIONAL ROSE ofrece algunas facilidades para instalar nuevos productos y extensiones en el entorno de la aplicación.

1.2 Creación y modificación de *scripts*

Para crear un nuevo *script* se realizan los siguientes pasos:

1. Seleccionar *Tools/New Script* del menú de ROSE.
2. Escribir el texto del *script*.
3. Seleccionar *File/Save As* del menú para salvar el nuevo *script*.

Y para modificar un *script* ya existente:

1. Seleccionar *Tools/Open Script* del menú de ROSE.
2. Seleccionar alguno de los *scripts* disponibles.
3. Pulsar *Ok* para entrar en el editor de *scripts* y visualizarlo.
4. Seleccionar el texto del *script* y copiarlo al portapapeles.
5. Seleccionar *Tools/New Script*.
6. Pegar el contenido del portapapeles en el nuevo *script*.
7. Seleccionar *File/Save As* para salvar el nuevo *script*.

1.3 Aspectos básicos del lenguaje de *scripts*

1.3.1 Inicio y finalización de *scripts*

Al igual que en muchos otros lenguajes, la ejecución del *script* comienza en la subrutina *Main*. Por lo tanto, todo *script* debe tenerla definida. Cuando la ejecución llega al *EndSub* correspondiente a la subrutina *Main* ésta se termina. El siguiente ejemplo muestra la definición habitual de *Main*:

```
Sub Main()
  MsgBox “Esta es la subrutina Main().”
End Sub
```

La sentencia *End* finaliza de manera forzada la ejecución del *script*, cerrando todos los ficheros abiertos. Este segundo ejemplo adelanta la finalización del *script* con la sentencia *End*:

```
Sub Main()
  MsgBox ”La próxima línea terminará la ejecución del script.”
  End
End Sub
```

1.3.2 Los comentarios

En el siguiente ejemplo se pueden observar varias formas de introducir comentarios en el código de un *script*:

```
  MsgBox “Hola”          'Visualiza un mensaje.

  REM Esto es un comentario.

  MsgBox “Antes del comentario”
  /* Este párrafo está todo comentado.
  Esta línea también será ignorada.
  Esta es la última línea del comentario. */
  MsgBox “Después del comentario”
```

1.3.3 Un aspecto sintáctico

El carácter de continuación de línea (_) permite dividir una sentencia compleja en varias líneas.

Capítulo 2

Introducción al lenguaje SUMMIT BASICSCRIPT

Como se ha dicho anteriormente, el lenguaje de *scripting* de RATIONAL ROSE está basado en una versión de BASIC, por lo que su aprendizaje resulta sencillo. A continuación se describen las principales características del lenguaje y que se podrán utilizar cuando se escriba un *script* para RATIONAL ROSE.

2.1 Tipos de datos simples

En primer lugar se muestran los principales tipos que ofrece el lenguaje.

2.1.1 Tipos lógicos

Boolean

Este tipo de datos representa a los valores lógicos $True$ y $False$. Por defecto, una variable booleana tendrá el valor $False$.

2.1.2 Tipos numéricos

Integer (%)

Este tipo de datos representa a los números enteros de hasta cuatro dígito de precisión en el siguiente rango:

$$-32768 \leq integer \leq 32767$$

Long (&)

Este tipo de datos representa a los números enteros largos de hasta 10 dígitos de precisión en el siguiente rango:

$$-2,147,483,648 \leq Long \leq 2,147,483,647$$

Currency (@)

Este tipo de datos representa a los números en punto–fijo con 15 dígitos a la izquierda del punto decimal y cuatro a la derecha, por lo que son útiles para cálculos monetarios. El rango es el siguiente:

$$-922,337,203,685,477.5808 \leq currency \leq 922,337,203,685,477.5807$$

Single (!)

Este tipo de datos representa a los números reales con 7 dígitos de precisión en los siguientes rangos:

$$-3.402823\,E^{38} \leq single \leq -1.401298\,E^{-45}$$
$$1.401298\,E^{-45} \leq single \leq 3.402823\,E^{38}$$

Double (#)

Este tipo de datos representa a los números reales con 15–16 dígitos de precisión en los siguientes rangos:

$$-1.797693134862315\,E^{308} \leq double \leq -4.94066\,E^{-324}$$
$$4.94066\,E^{-324} \leq double \leq 1.797693134862315\,E^{308}$$

2.1.3 Tipos de cadenas de caracteres

String ($)

Este tipo de datos representa a las cadenas de caracteres. Cada carácter tiene un valor entre 0 y 255 (el habitual código ASCII) y la longitud máxima de una cadena es de 32767 caracteres. Al contrario que en otros lenguajes, las cadenas pueden contener nulos, como se muestra en el siguiente ejemplo:

s$ = "¡"+"Hola"+Chr$(0)+"!"   'Esta cadena incluye un carácter nulo.

La función *Len* devuelve el número de caracteres de la cadena (incluyendo los caracteres no imprimibles). Normalmente una cadena se declara con longitud variable (la memoria requerida para su almacenamiento dependerá de su longitud). Las variables cadena no asignadas tienen por defecto una longitud 0. También se pueden declarar con un tamaño fijo (más apropiadas para el paso de estructuras con tamaño fijo a rutinas externas). Por ejemplo:

Dim s As String
s = "Hola"   'Esta cadena tiene longitud 4.

Dim s As String * 20
s = "Hola"   'Esta cadena tiene longitud 20 (el espacio sobrante es rellenado con espacios).

Al asignar una expresión (de tipo cadena) a una variable (también de tipo cadena) se aplican las siguientes reglas:

- Se rellena con espacios hasta alcanzar la longitud de la cadena (en caso de que fuera menor).
- Se trunca a la longitud de la cadena (en caso de que sea mayor).

2.1.4 Tipos de fechas y horas

Date

Este tipo de datos representa a las fechas y horas de manera conjunta. Su rango es el siguiente:

$$January\,1,100\ 00{:}00{:}00 \leq date \leq December\,31,9999\ 23{:}59{:}59$$

$$-6574340 \leq date \leq 2958465.99998843$$

En el formato numérico, la parte entera almacena el número de días desde el *31–Dic–1899*, y la parte fraccional el número de segundos transcurridos del día. El valor por defecto es 0. Por ejemplo:

```
Dim d As Date
d = #January 1, 1990#
```

Para evitar ambigüedades debidas a los diversos formatos utilizados en distintos paises se recomienda la utilización del formato de fecha universal:

```
VariableFecha = #YY/MM/DD HH:MM:SS#
```

2.1.5 Tipos variantes

Variant

Es el tipo por defecto en BasicScript y se utiliza para declarar variables que puedan contener datos de diferentes tipos. Durante la existencia de un *Variant* su tipo puede cambiar entre:

Numéricos *Integer*, *Long*, *Single*, *Double*, *Boolean*, *Currency*.

Lógico *Boolean*.

Temporal *Date*.

Cadena *String*.

Objecto *Object*.

Dato inválido Un *Variant* con datos inválidos se considera nulo (*Null*).

Dato sin iniciar Un *Variant* sin inicializar se considera vacío (*Empty*).

Las siguientes funciones se utilizan para determinar el tipo del dato contenido:

VarType Devuelve una constante que representa al tipo.

IsNumeric Devuelve $True$ si el tipo es numérico, una cadena que contiene un número o un objecto cuya propiedad por defecto sea numérica.

IsDate Devuelve $True$ si contiene una fecha, una cadena convertible a fecha o un objecto cuya propiedad por defecto sea una fecha.

IsObject Devuelve $True$ si contiene un objeto.

IsNull Devuelve $True$ si no contiene un dato válido.

IsEmpty Devuelve $True$ si no está inicializado.

Asignación a una variable de tipo variante

Inicialmente se le considera vacío ($VarType$ devolverá *ebEmpty*). Si no está inicializado devolverá 0 en las expresiones numéricas o una cadena de longitud cero en una expresión de cadenas. Si se quiere dar un valor nulo a un $Variant$ que ya está inicializado hay que asignarle otro $Variant$ que esté vacío. Cuando se le asigna un valor también se le asigna un tipo y por lo tanto en todas las operaciones posteriores se comportará como cualquier variable de ese tipo.

Operaciones sobre variables de tipo variante

Normalmente, un $Variant$ se comportará según el tipo de datos que contenga. La única excepción es la promoción automática de los tipos numéricos en caso de salidas de rangos. Por ejemplo:

```
Dim a As Integer, b As Integer, c As Integer
Dim x As Variant, y As Variant, z As Variant

a% = 32767
b% = 1
c% = a% + b% 'Esta operación se saldrá de rango
x = 32767
y = 1
z = x + y          'z se convierte a Long porque la suma está fuera del rango entero
```

Sumas con variables de tipo variante

Según el contenido de un $Variant$ el operador "+" puede realizar una de dos funciones: concatenar cadenas o sumar números. Debido a que el tipo no es conocido hasta el momento de la ejecución se pueden realizar operaciones erróneas. Se recomienda utilizar el operador "&" para concatenar dos $Variant$ que contengan cadenas.

Variables de tipo variante sin contenido

Una variable $Variant$ puede tener asignado un valor especial que indica que no contiene dato válido ($Null$). El valor $Null$ se puede utilizar para buscar errores porque este valor se propaga a través de las expresiones. Por ejemplo:

```
Dim a As Variant
a = Null
```

Desventajas de los tipos variantes

1. La evolución del programa es más lenta comparada con los tipos fundamentales porque cada operación implica el examen del tipo contenido.

2. Comparado con otros tipos de datos, requieren un mayor almacenamiento.
3. En ocasiones, el comportamiento de un *Variant* es impredecible debido a los cambios automáticos de tipo.

2.1.6 Otros tipos de datos simples

Any

Se utiliza en la definición de procedimientos y funciones cuando se quiere indicar que no se debe realizar el chequeo de tipo en el argumento (que está declarado como tal). Por ejemplo:

```
Declare Sub Foo Lib "FOO.DLL" (a As Any)

  ...

  ' Ambas llamadas son correctas
  Foo 10
  Foo "Hola a todos."
```

2.2 Tipos de datos especiales

En esta sección se muestran tres tipos de datos que por su importancia merecen un tratamiento un poco más profundo. Se remarcan de manera especial los objetos porque sobre ellos se basará la mayor parte del procesamiento en un *script*.

2.2.1 Arrays

Se declaran utilizando las sentencias *Dim*, *Public* o *Private*. Por ejemplo:

```
Dim a(10) As Integer
Public LastNames(1 to 5,-2 to 7) As Variant
Private ...
```

Se pueden crear arrays de cualquier tipo: *Integer*, *Long*, *Single*, *Double*, *Boolean*, *Date*, *Variant*, *Object* o estructuras definidas por el usuario. Los límites de cada dimensión del array deben estar dentro del rango de los enteros. Un array puede tener hasta 60 dimensiones. Se pueden declarar de dos tipos:

Arrays estáticos Sus dimensiones no pueden variar en tiempo de ejecución. Una vez declarado siempre requerirá la misma cantidad de almacenamiento. En la declaración sólo se diferencian porque se indican sus dimensiones. Son los únicos arrays que pueden aparecer dentro de una estructura. Por ejemplo:

```
Dim a(10) As String

Type Foo
  rect(4) As Integer
  colors(10) As Integer
End Type
```

Arrays dinámicos Se declaran sin unas dimensiones explícitas y pueden ser redeclarados en tiempo de ejecución (preservando o no el array anterior). No pueden ser miembros de los tipos definidos por el usuario. Por ejemplo:

```
Public Ages() As Integer
Redim Ages$(100)
Redim Preserve Ages$(100)
```

Funciones

LBound Límite inferior.

Ubound Límite superior.

ArrayDims Número de dimensiones.

Sentencias y subrutinas

ArraySort Ordena un array (su tipo debe ser *Integer*, *Long*, *Single*, *Double*, *Currency*, *Boolean*, *Date* o *Variant*).

FileList Rellena un array con la lista de ficheros de un directorio.

DiskDrives Rellena un array con la lista de letras para dispositivos válidos.

AppList Rellena un array con la lista de aplicaciones que se están ejecutando.

WinList Rellena un array con la lista de ventanas de primer nivel.

SelectBox Visualiza los contenidos de un array en un control de lista.

PopupMenu Visualiza los contenidos de un array en un menú emergente.

ReadInSection Rellena un array con los ítemes de una sección en un fichero INI.

FileDirs Rellena un array con una lista de subdirectorios.

Erase Borra todos los elementos de un array.

ReDim Restablece los límites y dimensiones de un array.

Dim Declara un array

2.2.2 Estructuras

Son estructuras definidas en el propio *script* utilizando la sentencia *Type* y son equivalentes a las estructuras del lenguaje C. Tienen un carácter global, por lo que se deben declarar fuera del cuerpo de las subrutinas y funciones de un *script*.

Cada miembro de una estructura puede ser de cualquier tipo de dato fundamental, incluyendo otras estructuras, con la excepción de los arrays de longitud variable. Una vez definida la estructura se podrá utilizar para declarar variables con las sentencias *Dim*, *Public* o *Private*. Por ejemplo:

```
Type Rect
  izq As Integer
  sup As Integer
  der As Integer
  inf As Integer
End Type
  ...
Sub Main()
  Dim r As Rect
    ...
  r.izq = 10
End Sub
```

Las variables que sean estructuras del mismo tipo pueden asignarse unas a otras (copiando sus contenidos). Ningún otro operador estándar puede utilizarse. Cuando se copian estructuras del mismo tipo todas las cadenas en la variable fuente son duplicadas y las referencias se colocan en la variable destino.

```
  Dim r1 As Rect
  Dim r2 As Rect
    ...
  r1 = r2
```

La sentencia *LSet* puede utilizarse para copiar variables cuyos tipos de estructuras sean diferentes. La menor de las estructuras determinan cuántos bytes serán copiados. Por otro lado, la función *Len* puede utilizarse para determinar el número de bytes que ocupa una estructura (si la estructura contiene cadenas esta función puede devolver información incorrecta porque la estructura sólo contiene las referencias a la cadenas).

2.2.3 Objetos

Object

Se pueden definir dos tipos de objetos en BasicScript: objetos de datos y objetos de automatización OLE. Sintácticamente, ambos son idénticos.

BasicScript considera un objeto como una encapsulación de datos y rutinas en una unidad simple. La utilización de objetos en BasicScript tiene el efecto de agrupar en un mismo conjunto funciones y datos que se aplican sólo a un tipo de objeto específico. Los objetos ofrecen sus datos a través de las propiedades. Normalmente, las propiedades pueden leerse (*get*) o modificarse (*set*). Los objetos también ofrecen sus rutinas internas a través de los métodos. Cada método de objeto puede tomar la forma de una función o una subrutina.

Principalmente, el tipo *Object* se utiliza para declarar variables que hacen referencia a objetos dentro de una aplicación mediante la automatización OLE. En realidad, cada variable de este tipo es un puntero de 4 bytes que hace referencia al objeto físico real en cuestión. El valor 0 (*Nothing*) indica que la variable no hace referencia a un objeto válido (por lo tanto, no se puede acceder a sus propiedes ni métodos porque se produciría un error en tiempo de ejecución). Se declaran utilizando las sentencias *Dim*, *Public* o *Private*:

```
  Dim MiApl As Object
```

A estas variables se les asignan valores utilizando la sentencia *Set*:

```
Set MiApl = CreateObject("NombreAplicacion")
Set MiApl = Nothing
```

Se utiliza el separador *punto* para acceder a las propiedades y métodos de una variable *Object*:

```
MiApl.Color = 10
i% = MiApl.Color
...
MiApl.Open "Ejemplo.txt"
tuvoExito = MiApl.Save("Nuevo.txt",15)
```

En general, se pueden realizar dos tipos de comparaciones:

- Para determinar si se refieren al mismo objeto.
- Para determinar si la variable objeto referencia o no un objeto válido (comparando con *Nothing*).

Las comparaciones entre objetos se realizan mediante el operador *Is*:

```
If a Is b Then MsgBox "a y b no son el mismo objeto."
If a Is Nothing Then MsgBox "a no está inicializado."
If b Is Not Nothing Then MsgBox "b está en uso."
```

BasicScript mantiene el número de referencias a un objeto, por lo que el objeto sólo será destruido cuando no haya más referencias al mismo:

```
Sub Main()                                    'Número de referencias al objeto
  Dim a As Object                             '0
  Dim b As Object                             '0
  Set a = CreateObject("NombreAplicacion")    '1
  Set b = a                                   '2
  Set a = Nothing                             '1
End Sub                                       'Objeto destruido
```

Creación de objetos OLE

La función *CreateObject* crea un objeto de automatización OLE y devuelve una referencia a ese objeto. El parámetro *class* especifica la aplicación utilizada para crear el objeto y el tipo de objeto que se está creando, siguiendo la siguiente sintáxis: *"application.class"*.

En tiempo de ejecución, *CreateObject* busca la aplicación y la ejecuta. Cuando el objeto se ha creado, se puede acceder a sus propiedades y métodos utilizando la sintáxis habitual, por ejemplo: *object.property = value*. El acceso a aplicaciones OLE puede provocar cierto enlentecimiento en la ejecución del *script*.

En un primer ejemplo se instancia el programa Microsoft Excel, utilizando entonces el objeto resultante para hacerlo visible y cerrarlo:

```
Sub Main()
  Dim Excel As Object
  On Error GoTo Punto1                           'Establecer captura de errores.
  Set Excel = CreateObject("Excel.Application") 'Instaciar el objeto.
  Excel.Visible = True                           'Visualizar Excel.
  Sleep 5000                                     'Esperar 5 segundos.
  Excel.Quit                                     'Cerrar Excel.
  Exit Sub                                       'Salida previa al gestor de errores.

Punto1:
  MsgBox "No se pudo crear el objeto Excel."  'Visualizar un mensaje de error.
  Exit Sub                                    'Reiniciar el gestor de errores.
End Sub
```

En un segundo ejemplo se utiliza la función para instanciar un objeto VISIO que será utilizado para crear un nuevo documento:

```
Sub Main()
  Dim Visio As Object
  Dim doc As Object
  Dim pagina As Object
  Dim forma As Object

  Set Visio = CreateObject("Visio.Application")  'Crear un objeto Visio.
  Set doc = Visio.Documents.Add("")              'Crear un nuevo documento.
  Set pagina = doc.Pages(1)                      'Obtener la primera página.
  Set forma = pagina.DrawRectangle(1,1,4,4)

  forma.text = "¡Hola mundo!"                    'Escribir un texto dentro de la forma.
End Sub
```

Asignación/Inicialización de objetos OLE

La función *GetObject* se utiliza para obtener un objeto de automatización OLE existente. Su sintáxis es:

$$GetObject(pathname[, class])$$

El parámetro *pathname* especifica el camino completo del fichero que contiene el objeto a ser activado. La aplicación asociada con el fichero es determinada por OLE en tiempo de ejecución. Por ejemplo, supóngase un fichero *c:\docs\resumen.doc* que fue creado con el procesador de textos *wordproc.exe*. La siguiente sentencia invocaría este programa, cargaría el fichero y asignaría ese objeto a una variable:

```
  Dim doc As Object
  Set doc = GetObject("c:\docs\resumen.doc")
```

Para activar una parte de un objeto, se añade una cadena que lo representa, separado por el operador "!". Por ejemplo, para activar las primeras tres páginas del documento del ejemplo anterior:

```
Dim doc As Object
Set doc = GetObject("c:\docs\resumen.doc!P1--P3")
```

Según los parámetros pasados a la función, ésta se comportará de una manera u otra:

- Si *pathname* no está especificado y sí lo está *class*, entonces *GetObject* devuelve una referencia a una instancia que ya existe del objeto especificado.
- Si *pathname* es " " y *class* está especificado, entonces *GetObject* devuelve una referencia a un nuevo objeto (de la clase especificada). Es equivalente a *CreateObject*.
- Si *pathname* está especificado pero no lo está *class*, entonces *GetObject* devuelve el objeto por defecto desde *pathname*. La aplicación que se debe activar es determinada por OLE según el nombre del fichero dado.
- Si tanto *pathname* como *class* están especificados, entonces *GetObject* devuelve un objeto según los parámetros especificados.

Algunos ejemplos:

```
'Este primer ejemplo devuelve una nueva referencia a la instancia existente de Excel.
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")

'Este segundo ejemplo carga el servidor OLE asociado con un documento.
Dim MiObj As Object
Set MiObj = GetObject("c:\docs\resumen.doc",)
```

Los objetos predefinidos

BasicScript tiene predefinidos unos cuantos objetos para ser utilizados en todos los *scripts*. No todos se encuentran disponibles en las diversas plataformas. Son:

- *Clipboard*
- *System*
- *Desktop*
- *HWND*
- *Net*
- *Basic*
- *Screen*

2.2.4 Colecciones

Una colección es un conjunto de variables objetos relacionadas. Cada elemento en el conjunto se denomina miembro y podemos acceder a él mediante un índice (numérico o textual). Es típico que los índices de las colecciones comiencen por 0. Por ejemplo:

```
MyApp.Toolbar.Buttons(0)
MyApp.Toolbar.Buttons("Tuesday")
```

Cada elemento de una colección es en sí mismo un objeto. Por ejemplo:

```
Dim MyToolbarButton As Object
Set MyToolbarButton = MyApp.Toolbar.Buttons("Save")
MyAppp.Toolbar.Buttons(1).Caption = "Open"
```

Una colección tiene propiedades para obtener información sobre ella y métodos que permiten la navegación por sus miembros. Por ejemplo:

```
Dim MyToolbarButton As Object
NumButtons% = MyApp.Toolbar.Buttons.Count
MyApp.Toolbar.Buttons.MoveNext
MyApp.Toolbar.Buttons.FindNext "Save"

For i = 1 To MyApp.Toolbar.Buttons.Count
  Set MyToolbarButton = MyApp.Toolbar.Buttons(i)
  MyToolbarButton.Caption = "Copy"
Next i
```

Propiedades de una colección

Count es la única propiedad aplicable a las colecciones. Es el número de objetos contenidos en la colección que permite la utilización de un contador para recorrer la colección mediante una iteración.

Métodos de las colecciones

A continuación se describen los métodos de una colección, que permiten la localización y procesamiento de elementos en ella. Aunque todas las propiedades y métodos sean los mismos, actúan sobre tipos diferentes de objetos según el tipo de objeto contenido. Por ejemplo, el método *ClassCollection.GetAt* obtiene una objeto "clase", y el método *CategoryCollection.GetAt* obtiene un objeto "categoría", etc. Estos métodos son:

Exists Indica si un objeto existe en una colección.

FindFirst Obtiene el índice (posición) de la primera instancia de un objeto en una colección.

FindNext Obtiene el índice (posición) de la siguiente instancia de un objeto en una colección.

GetWithUniqueID Obtiene la instancia de un objeto en una colección, dado su identificador *ID* único, excepto para aquellos tipos de objetos que no tienen *ID* único (como los objetos *ExternalDocument* o *Property*).

GetAt Obtiene una instancia específica de un objeto en una colección.

GetFirst Obtiene la primera instancia de un objeto en una colección.

GetObject Obtiene el interfaz OLE asociado con una colección dada.

IndexOf Obtiene el índice (posición) de un objeto en una colección dada.

Métodos para las colecciones definidas por el usuario

A continuación se describen cuatro métodos de colecciones adicionales, que permiten añadir o quitar objetos de una colección. Sin embargo, estos métodos sólo son válidos para aquellas colecciones definidas por el usuario y no pueden ser utilizadas en las colecciones correspondientes de un modelo de RATIONAL ROSE.

Add Añade un objeto a la colección de objetos.

AddCollection Añade una colección a una colección de objetos.

Remove Elimina una colección de una colección de objetos.

RemoveAll Elimina el contenido completo de una colección.

2.3 Constantes, variables y expresiones

2.3.1 Definición de constantes

Se puede declarar una constante para ser utilizada en el *script*. En el momento de la declaración, podrá asignarse una expresión formada por literales y otras constantes (no se permiten las llamadas a función, excepto $Chr\$$). Por ejemplo:

```
Const s$ = "Hola" + Chr(44)

Const a% = 5      'Constante entera cuyo valor es 5
Const b# = 5      'Constante real doble cuyo valor es 5.0
Const c$ = "5"    'Constante cadena cuyo valor es "5"
Const d! = 5      'Constante real simple cuyo valor es 5.0
Const e& = 5      'Constante entera larga cuyo valor es 5

Const a As Integer = 5    'Constante entera cuyo valor es 5
Const b As Double = 5     'Constante real doble cuyo valor es 5.0
Const c As String = "5"   'Constante cadena cuyo valor es "5"
Const d As Single = 5     'Constante real simple cuyo valor es 5.0
Const e As Long = 5       'Constante entera larga cuyo valor es 5

Const a = 5          'Constante entera
Const b = 5.5        'Constante real simple
Const c = 5.5E200    'Constante real doble

Const FichDef = "Defecto.txt"

Sub Test1
  Const FichDef = "Fichero.txt"
  MsgBox FichDef              'Visualiza "Fichero.txt".
End Sub

Sub Test2
  MsgBox FichDef              'Visualiza "Defecto.txt".
End Sub
```

En el siguiente ejemplo se muestran las constantes declaradas en una ventana de mensaje (*crlf* produce un salto de línea en la ventana):

```
Const crlf = Chr$(13) + Chr$(10)
Const s As String = “Esto es una constante.”

Sub Main()
  MsgBox s$ & crlf & “Las constantes se muestran arriba.”
End Sub
```

2.3.2 Declaración de variables

La sentencia *Dim*

La sentencia *Dim* declara una lista de variables locales con sus correspondientes tipos y tamaños. Su sintáxis es:

Dim *Name* [(< *Subscripts* >)] [As [New] *Type*] [,Name [(< *Subscripts* >)] [As [New] *Type*]] ...

Una alternativa a *Dim* es el *carácter de declaración de tipos*. Por ejemplo, las siguientes declaraciones son equivalentes:

```
  Dim Temperatura As Integer
  Dim Temperatura%
```

El parámetro *Subscripts* permite la declaración de arrays dinámicos y fijos (si se expresan los límites superiores e inferiores hasta 60 dimensiones) siguiendo la siguiente sintáxis:

[*Lower* To] *Upper* [,[*Lower* To] *Upper*]...

El parámetro *Type* especifica el tipo del dato que se declara y puede ser: *String*, *Integer*, *Long*, *Single*, *Double*, *Currency*, *Object*, objectos de datos o tipos de datos (del sistema o definidos por el usuario).

Una sentencia *Dim* dentro de una subrutina o función declara variables locales a ella. Si aparece fuera, entonces su alcance es global y equivalente al de las variables declaradas con la sentencia *Private*.

Cadenas de longitud fija

Se declaran añadiendo la longitud a la declaración del tipo *String*:

Dim nombre_variable As String * longitud_cadena

Declaración de variables implícitas

Cuando BasicScript encuentra una variable no declarada con *Dim*, entonces su tipo será el especificado por el carácter de declaración de tipos. Si no lo tiene, entonces la primera letra del nombre de la variable es buscado en las sentencias *DefType* para asignarle un tipo por defecto. Si aún así todavía no se ha encontrado un tipo para la variable, entonces será un *Variant*.

Declaración de objetos de automatización OLE

Se pueden declarar variables de un tipo de objeto explícito para objetos conocidos por BASICSCRIPT a través de las librerías de tipos. Se sigue la siguiente sintáxis:

Dim *NombreVariable* As *Application.Class*

El parámetro *Application* especifica la aplicación utilizada para registrar el objeto de automatización OLE y el parámetro *Class* especifica el tipo de objeto definido en la librería de tipo. De esta manera se está utilizando el enlace estático en tiempo de compilación y mejorando el rendimiento de la invocación a sus métodos.

Creación de nuevos objetos

La palabra clave opcional *New* se utiliza para declarar una nueva instancia del objeto de datos especificado y no se puede utilizar cuando se declaran arrays u objetos de automatización OLE. En tiempo de ejecución, se notificará a la aplicación que se está definiendo un nuevo objeto: la aplicación responderá mediante la creación de un nuevo objeto físico (dentro del contexto apropiado) y devolverá una referencia al mismo, que se asignará inmediatamente a la variable que se está declarando. Cuando se acabe el ámbito de la variable entonces se notificará a la aplicación para que ésta destruya ese objeto físico.

Convenciones para los nombres de variables

Los nombres de variables deben seguir las siguientes reglas:

1. Deben comenzar con una letra.

2. Pueden contener letras, dígitos y el carácter de subrayado (_). Las puntuaciones no se permiten. El signo de exclamación (!) se permite, pero si está en la última posición se interpretará como un carácter de declaración de tipos.

3. El último carácter del nombre puede ser un carácter de declaración de tipo: #, @, %, !, & y $.

4. No deben exceder los 80 caracteres de longitud.

5. No pueden contener palabras reservadas.

Ejemplos

Los siguientes ejemplos utilizan la sentencia *Dim* para declarar variables tipadas:

```
Sub Main()
  Dim i As Integer
  Dim l&                        'Entero largo
  Dim s As Single
  Dim d#                        'Real doble
  Dim c$                        'Cadena
  Dim MyArray(10) As Integer    'Array de 10 enteros
  Dim MyStrings$(2,10)          'Array de bidimensional de cadenas
  Dim Filenames$(5 to 10)       'Array de 6 cadenas (numeradas de 5 a 10)
  Dim Values(1 to 10, 100 to 200) 'Array de variantes bidimensional
End Sub
```

La sentencia *Public*

Declara una lista de variables públicas, es decir, variables que son globales para todas las subrutinas y funciones de todos los *scripts*. La sintaxis que se sigue es:

Public *Name* [(*Subscripts*)] [As *Type*] [,*Name* [(*Subscripts*)] [As *Type*]]...

Cuando no se especifica otra cosa, cualquier variable se declara implícitamente local a la rutina que la utiliza. La sentencia *Global* tiene el mismo significado que *Public*.

Cuando se utilizan variables compartidas, se debe asegurar que las declaraciones son idénticas en todos los *scripts*.

El siguiente ejemplo utiliza una subrutina para calcular el área de 10 círculos y visualiza el resultado en una ventana. Las variables R y Ar son declaradas como públicas para que puedan utilizarse tanto en $Main$ como en $Area$.

```
Const crlf = Chr$(13) + Chr$(10)

Public x#, ar#

Sub Area()
  ar# = (x# ^ 2) * Pi
End Sub

Sub Main()
  msg = "Las áreas de los diez círculos son:" & crlf

  For x# = 1 To 10
    Area
    msg = msg & x# & ": " & ar# & Basic.Eoln$
  Next x#

  MsgBox msg
End Sub
```

La sentencia *Private*

Declara una lista de variables privadas, es decir, variables que son globales para todas las subrutinas y funciones que se ejecuten en el *script* actual. La sintáxis que se sigue es:

Private *Name* [(*Subscripts*)] [As *Type*] [,*Name* [(*Subscripts*)] [As *Type*]]...

La sentencia *Redim*

Se utiliza para volver a declarar un array, mediante la especificación de un límite inferior y otro superior para cada dimensión del array. La sintáxis de esta sentencia es:

Redim [Preserve] *VariableName* ([*SubscriptRange*]) [As *Type*], ...

Si la variable ya estaba anteriormente definida, entonces debe haberse declarado previamente mediante la sentencia *Dim* como un array dinámico (sin especificar dimensiones). Estos arrays pueden ser re–dimensionados cualquier número de veces.

El parámetro *SubscriptRange* especifica las nuevas dimensiones, y cuando no se especifican, se considera que el array no tendrá elementos. Por defecto, el límite inferior es cero, aunque depende de la sentencia *Option Base*. Se sigue la sintáxis:

[*Lower* To] *Upper* [,[*Lower* To] *Upper*] ...

Las dimensiones del nuevo array deben estar en el siguiente rango, generándose un error en caso de que no se cumplan:

$$-32768 \leq Lower \leq Upper \leq 32767$$

El parámetro *Type* puede utilizarse para especificar el tipo de cada elemento del array. Estos están restringidos a cualquier tipo de datos fundamental, tipos de datos definidos por el usuario o objetos.

La redimensión del array borra todos los elementos al menos que no se utilice la palabra clave *Preserve*. En este caso, los datos del array se mantienen si ello es posible. Si el número de elementos se incrementa, los nuevos elementos se inicializan a cero o cadena vacía. Si el nuevo tamaño es inferior, los elementos sobrantes serán borrados. Cuando se indica *Preserve*, el nuevo número de dimensiones deberá ser cero o el mismo número que ya tenía.

El siguiente ejemplo utiliza la sentencia *FileList* para redimensionar un array y rellenarlo con cadenas de nombres de ficheros. Un nuevo array es entonces redimensionado para guardar tantos elementos encontrados por *FileList*, copiando el array y visualizándolo parcialmente:

```
Sub Main()
  Dim fl$()

  FileList fl$,"*.*"

  contador = Ubound(fl$)
  Redim nl$(Lbound(fl$) To Ubound(fl$))

  For x = 1 to contador
    nl$(x) = fl(x)
  Next x

  MsgBox "El último elemento del nuevo array es : " & nl$(contador)
End Sub
```

2.3.3 Expresiones

Como en cualquier lenguaje, Las expresiones están compuestas por operadores y operandos. BasicScript permite utilizar variables, llamadas a funciones, constantes y literales con diferentes tipos de datos en las expresiones. Cuando esto ocurre, los dos argumentos se convierten al tipo del operando con mayor precisión. Si una operación se realiza entre una expresión numérica y otra textual, entonces la última se convierte a una expresión numérica.

Cuando una variable de tipo objeto es utilizada con operadores numéricos tales como la suma o la resta, se utiliza automáticamente la *propiedad por defecto del objeto.* En el siguiente ejemplo, las dos operaciones posteriores a la declaración del objeto son equivalentes porque la propiedad por defecto de un objeto de la clase *Excel* es *.Value*:

```
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")

MsgBox "Esta aplicación es " & Excel
MsgBox "Esta aplicación es " & Excel.Value
```

Operadores

A continuación se definen los operadores existentes en BasicScript. Hay que recordar que según los tipos de los operandos variará el resultado de la operación.

Concatenación de cadenas &

División entera \

Suma, resta, multiplicación y división $+$, $-$, $*$, $/$

Potencia ˆ

Comparaciones $<$, $\leq$, $>$, $\geq$, $=$, $<>$

Conjunción lógica o binaria *And*

Equivalencia lógica o binaria *Eqv* (devuelve *True* si ambos operandos tienen el mismo valor lógico)

Implicación lógica o binaria *Imp* (devuelve *True* si el segundo operando es *True*)

Identificación de clase *Is* (devuelve *True* si las dos variables objetos que son operandos hacen referencia al mismo objeto. Esta comparación puede realizarse con *Nothing* para determinar si la variable objeto está sin inicializar)

Comparación con patrones *Like*

Resto de una división *Mod*

Negación lógica o binaria *Not*

Disyunción lógica o binaria *Or*

Exclusión lógica o binaria *Xor* (devuelve *True* si ambos operandos tienen distintos valores lógicos)

Precedencia de operadores

A continuación se muestra la precedencia entre operadores en BasicScript (de mayor a menor precedencia). A igual precedencia, se evalúan de izquierda a derecha.

Paréntesis ()

Exponenciación ˆ

Menos unario $-$

División y multiplicación $/ \ *$

División entera $\backslash$

Módulo Mod

Suma y resta $+ \ -$

Concatenación de cadenas $\&$

Comparativas $= \ <> \ > \ < \ <= \ >=$

Comparación de cadenas y objetos $Like \ Is$

Negación lógica Not

Conjunción lógica o binaria And

Disjunción lógica o binaria Or

Operadores lógicos o binarios $Xor \ Eqv \ Imp$

Precisión de los operandos

BasicScript realiza la conversión de tipos automáticamente, pero algunas veces esto puede ocasionar algún error de rango. Por ejemplo, cuando los valores de punto flotante se convierten a enteros, la parte fraccional se pierde, redondeando al entero más cercano utilizando el redondeo de Baker:

- Si la parte fraccional es mayor que .5, el número es redondeado hacia arriba.
- Si la parte fraccional es menor que .5, el número es redondeado hacia abajo.
- Si la parte fraccional es igual a .5, el número es redondeado hacia arriba si es impar y hacia abajo si es par.

Cuando se utilizan operadores numéricos, binarios, lógicos o comparativos, el tipo de dato del resultado es generalmente el mismo del tipo de datos del operando más preciso. Por ejemplo, al sumar un entero y un entero largo, primero se convierte el entero a un entero largo, y después se realiza la suma, saliéndose de rango sólo cuando el resultado no puede contenerse en un entero largo. El orden de precisión se muestra en la siguiente lista (de menor a mayor precisión):

- $Empty$
- $Boolean$
- $Integer$
- $Long$
- $Single$
- $Date$
- $Double$
- $Currency$

Hay algunas excepciones con algunos operadores. Las reglas para la conversión de operandos son más complicadas cuando algún operando es de tipo $Variant$. En muchos casos, un valor fuera de rango causa la promoción automática del resultado al siguiente tipo de datos más preciso.

Literales

Los literales son valores con un tipo específico implícito que se pueden utilizar en las expresiones. Algunos ejemplos de literales son:

- 10
- 43265
- 5#
- 5.5
- $5.4E100$
- $\&HFF$
- $\&O47$
- $\&HFF\#$
- “Hola”
- ““Hola””
- #1/1/1994#

2.4 Funciones y subrutinas

2.4.1 Declaración de prototipos de subrutinas

La sentencia *Declare* crea el prototipo de una rutina externa o una rutina de BasicScript que se definirá posteriormente en el módulo fuente o en otro módulo. La sentencia *Declare* debe aparecer en el exterior de cualquier declaración *Sub* o *Function* y el prototipo declarado sólo será válido en aquel *script* donde aparezca. Sigue la siguiente sintáxis:

Declare {Sub | Function} $Name$[$TypeChar$] [CDecl | Pascal | System | StdCall]
    [Lib “LibName$” [Alias “AliasName$”]]
    [([$ParameterList$])] [As $Type$]

$ParameterList$ es la lista de parámetros (hasta 30) de la rutina y cada uno de ellos tiene la siguiente sintáxis:

[Optional] [ByVal | ByRef] $ParameterName$[()] [As $ParameterType$]

Los parámetros que se utilizan son los siguientes:

- **Name [TypeChar]** Nombre de la rutina. En las funciones, es opcional indicar el tipo de valor a devolver colocando un carácter de declaración de tipos (cualquiera de ellos excepto @ para el caso de funciones externas).

- **LibName$** Debe ser especificada si la rutina es externa para indicar el nombre de la librería o código fuente que la contiene y aparecerá entre comillas. Pueden utilizarse alias en caso de conflictos (**AliasName$** debe aparecer entre comillas). Por ejemplo:
  - *Declare Function GetCurrentTime Lib "user" () As Integer*
  - *Declare Function GetTime Lib "user" Alias "GetCurrentTime" As Integer*
- **Type** Indica el tipo a devolver. Para las funciones externas, los tipos válidos son: *Integer*, *Long*, *String*, *Single*, *Double*, *Date*, *Boolean* y objetos de datos. Por otro lado, los tipos *Currency* y *Variant*, las cadenas de longitud variable, los arrays, los tipos definidos por el usuario y los objetos de automatización OLE no pueden ser devueltos por las funciones externas.
- **Optional** Para indicar la opcionalidad de un parámetro. Todos los parámetros opcionales deben ser de tipo *Variant*, y se declaran al final de la lista de parámetros.
- **ByVal** Se pasará el parámetro por valor, por lo que no podrá ser modificados por la rutina.
- **ByRef** Se pasará el parámetro por referencia, por lo que podrá cambiarse. Por defecto, todos los parámetros se pasan por referencia.
- **ParameterName** Nombre del parámetro, que debe seguir las convenciones de nombres de BasicScript. "()″ indica que el parámetro es un array.
- **ParameterType** Especifica el tipo del parámetro. La cláusula **As** deber ser incluida sólo si el nombre del parámetro no contiene un carácter de declaración de tipo. Además de los tipos básicos, pueden especificarse estructuras definidas por el usuario, objetos de datos y objetos de automatización OLE. Para pasar parámetros a rutinas externas, si el tipo de dato no es conocido, se puede utilizar **Any**, forzando al compilador a relajar el chequeo de tipo, permitiendo que cualquier tipo de datos pueda ser pasado en el lugar de dicho argumento. Por ejemplo:
  - *Declare Sub Convertir Lib "MiLibreria" (a As Any)*

Paso de parámetros

Por defecto, el paso de parámetros en BasicScript es por referencia. Muchas rutinas externas requieren un valor en lugar de una referencia y para ello está *ByVal*. Por ejemplo, el procedimiento en C $void\ MessageBeep(int)$ debería declararse así:

```
Declare Sub MessageBeep Lib "user" (ByVal n As Integer)
```

El siguiente procedimiento en C $int\ SystemParametersInfo(int, int, int*, int)$ es un ejemplo de paso de parámetros por referencia y se escribiría así:

```
Declare Function SystemParametersInfo Lib "user" _
    (ByVal action As Integer, _
     ByVal uParam As Integer, _
     ByRef pInfo As Integer, _
     ByVal updateINI As Integer) As Integer
```

Paso de cadenas

Las cadenas pueden pasarse por referencia o por valor. Cuando se pasan por referencia, lo que se pasa es un puntero a un puntero a una cadena al estilo C (terminada con *null*) y la rutina externa puede cambiar el puntero o modificar el contenido de la cadena. Cuando se pasa por valor, entonces se pasa un puntero a una cadena al estilo C. En ambos casos, si una rutina externa modifica la cadena que se le pasa, debe haber suficiente espacio para almacenar los caracteres devueltos y para ello podemos utilizar la función *Space*. Por ejemplo, una función que llama a una DLL de 16–bit de Windows:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal dirname$, ByVal length%)

Sub Main()
  Dim s As String
  s = Space(128)
  GetWindowsDirectory s,128
End Sub
```

Una alternativa para asegurar que una cadena tiene suficiente espacio es declararla con una longitud fija:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal dirname$, ByVal length%)

Sub Main
  Dim s As String * 128
  GetWindowsDirectory s,len(s)
End Sub
```

Convenciones de llamada a rutinas externas

La lista de argumentos debe encajar exactamente en la rutina externa. Cuando se llama a una de ellas BasicScript necesita información sobre cómo espera la rutina recibir sus parámetros y quién es responsable de limpiar la pila. Existen distintas convenciones de llamadas que son dependientes de cada plataforma:

StdCall *_stdcall* (convención de llamada estándar) Los argumentos se introducen en la pila de derecha a izquierda y la función llamada limpia la pila.

Pascal *pascal* (convención de llamada de Pascal) Los argumentos se introducen en la pila de izquierda a derecha y la función llamada limpia la pila.

System *_System* (convención de llamada del sistema) Los argumentos se introducen en la pila de derecha a izquierda y la función que llama limpia la pila. El número de argumentos se especifica en el registro *AL*.

CDecl *cdecl* (convención de llamada de C) Los argumentos se introducen en la pila de derecha a izquierda y la función que llama limpia la pila.

Paso de punteros nulos

Existen varias formas para pasar cadenas no inicializadas a un rutina externa preparada para recibir cadenas por valor. La primera de ella es pasar una constante *ebNullString*, por ejemplo:

```
Declare Sub EjProc Lib "Ejemplo" (ByVal lpNombre As Any)

Sub Main()
  EjProc ebNullString        'Pasa un puntero nulo
End Sub
```

Otra manera de pasar una cadena no inicializada es declarar el parámetro que recibirá el puntero nulo como *Any*, y pasarle un 0 por valor:

```
Declare Sub EjProc Lib "Ejemplo" (ByVal lpNombre As Any)

Sub Main()
  EjProc ByVal 0&        'Pasa un puntero nulo
End Sub
```

Paso de datos a rutinas externas

Cuando se pasa un objeto, tanto por valor como por referencia, sólo podrá ser usado por rutinas externas escritas específicamente para BasicScript. Si es un objeto de automatización OLE se pasa un puntero largo a un manejador *LPDISPATCH*.

Los arrays sólo se pueden pasar por referencia. Las estructuras no tienen ningún otro problema, excepto que el alineamiento es cada 2 bytes, que no es consistente con C. Los diálogos no pueden pasarse a las rutinas externas. Sólo pueden pasarse cadenas de longitud variable (las fijas se convierten automáticamente a variables).

BasicScript pasa los datos a las funciones externas de manera consistente con su prototipo (definido con *Declare*). Hay una excepción a esta regla: se puede pasar un parámetro por valor aunque esté definido por referencia (utilizando *ByVal* en la llamada). Por ejemplo:

```
Declare Sub EjProc Lib "MiLib" (ByRef i As Integer)

Sub Main
  Dim i As Integer
  i = 6
  EjProc 6          'Pasa un entero temporal (de valor 6) por referencia
  EjProc i          'Pasa la variable "i" por referencia
  EjProc (i)        'Pasa un entero temporal (de valor 6) por referencia
  EjProc i + 1      'Pasa un entero temporal (de valor 7) por referencia
  EjProc ByVal i    'Pasa "i" por valor
End Sub
```

Hay que tener cuidado con este uso de *ByVal*. Por ejemplo, el procedimiento *EjProc* anterior espera recibir un puntero a entero de 32 bits, pero se fuerza el paso de un entero por valor de 16 bits. Esta diferencia de tamaños puede dar lugar a resultados impredecibles.

Devolución de valores desde rutinas externas

BasicScript permite los siguientes tipos: *Integer*, *Long*, *Single*, *Double*, *String*, *Boolean* y todos los tipos objetos. Cuando se devuelve una cadena, se asume que el primer carácter nulo será el final de la cadena.

Llamada a rutinas externas en entornos multitareas

En estos entornos, BasicScript hace una copia de todos los datos pasados a las rutinas externas. Esto permite que otros *scripts* que se ejecuten simultaneamente continuen su ejecución antes de que la rutina externa termine. Hay que tener cuidado cuando se pase la misma variable por referencia dos veces a las rutinas externas. Cuando vuelva de estas llamadas, BasicScript debe modificar el dato real de las copias hechas antes de la llamada, y como la misma variable se pasó dos veces, sera imposible determinar qué variable se modifica. Por ejemplo:

```
Declare Function IsLoaded% Lib "Kernel" Alias "GetModuleHandle" (ByVal name$)

Declare Function GetProfileString Lib "Kernel"
   (ByVal SName$, _
    ByVal KName$, _
    ByVal Def$, _
    ByVal Ret$, _
    ByVal Size%) As Integer

Sub Main()
  SName$ = "Intl"               'Nombre de la sección de Win.ini
  KName$ = "Country"            'La variable para el país en Win.ini
  ret$ = String$(255, 0)        'Inicializa la cadena a devolver.
  If GetProfileString(SName$,KName$,"",ret$,Len(ret$)) Then
    MsgBox "El país configurado es : " & ret$
  Else
    MsgBox "No hay configuración del país en el fichero Win.ini"
  End If

  If IsLoaded("Progman") Then
    MsgBox "Progman está cargado."
  Else
    MsgBox "Progman no está cargado."
  End If
End Sub
```

El uso de *Declare* para la referencia a rutinas externas es dependiente de cada plataforma, por lo que se deben revisar las características especiales para cada una de ellas.

2.4.2 Declaración de funciones

Para crear una función definida por el usuario se sigue la siguiente sintáxis:

[Private | Public] [Static] Function *Name*[(*Arglist*)] [As *ReturnType*]
  [*Statements*]
End Sub

La lista de argumentos es una lista separada por comas de la definición de hasta 30 parámetros (número máximo permitido) que sigue la siguiente sintáxis:

[Optional] [ByVal | ByRef] *Parameter* [()] [As *Type*]

La sentencia *Function* tiene las siguientes partes:

Private Indica que la función no puede llamarse desde otros *scripts*.

Public Indica que la función puede llamarse desde otros *scripts* (opción por defecto).

Static Aunque el compilador reconoce esta opción, no tiene efecto alguno.

Name Nombre de la función (sigue las mismas reglas que los nombres de variables).

Optional Indica que el parámetro es opcional. Todos ellos deben ser del tipo $Variant$ y todos los que le siguen se consideran también opcionales. La función $IsMissing$ permite saber si se pasó o no un determinado parámetro opcional.

ByVal Indica que el parámetro se pasa por valor.

ByRef Indica que el parámetro se pasa por referencia (opción por defecto).

Parameter Nombre del parámetro (sigue las mismas reglas que los nombres de variables). Pueden llevar un carácter de declaración de tipo (entonces no es necesario poner la cláusula As).

Type Tipo del parámetro. Los arrays se indican con paréntesis, por ejemplo: *a() As Integer.*

ReturnType Tipo del dato devuelto por la función (por defecto es $Variant$). También se puede especificar con un carácter de declaración de tipo en el nombre de la función.

Una función retorna cuando se encuentra alguna de las siguientes sentencias: *End Function* o *Exit Function.* Las funciones pueden ser recursivas.

Devolviendo valores desde las funciones

Para asignar un valor de retorno, se asigna una expresión al propio nombre de la función. Por ejemplo:

```
Function Doblar(a As Integer) As Integer
  Doblar = a * 2
End Function
```

Si no se asigna algún valor antes de la finalización de la función, entonces, según el tipo, se devolverá: 0 (tipos numéricos), una cadena de longitud 0 (tipos cadenas), $Nothing$ (cualquier tipo de objeto), $Error$ (los tipos $Variant$), *December* 30, 1899 (los tipos de fechas) o $False$ (tipo lógico).

Algunos ejemplos de declaraciones equivalente de funciones respecto al tipo a devolver :

```
Function Test() As String
  Test = "Hola mundo"
End Function

Function Test$()
  Test = "Hola mundo"
End Function
```

El paso de parámetros a las funciones

Pueden pasarse por valor o por referencia, dependiendo de su declaración. Los efectos son conocidos: si una variable se pasa por valor, entonces su contenido no puede cambiarse; al contrario, si se pasa por referencia, cualquier modificación sobre su contenido se ve reflejada cuando se retorne al punto de llamada.

A pesar de la declaración, se puede forzar el tipo de paso del parámetro por referencia en la llamada si se pone entre paréntesis el nombre de la variable actual. Por ejemplo, en la siguiente función, se pasa j por referencia aunque el paso de parámetro se haya declarado por valor:

$$i = FuncionUsuario(10, 12, (j))$$

Parámetros opcionales

Se permite que en la llamada a funciones, algunos parámetros sean opcionales, como se muestra en el siguiente ejemplo:

```
Function Test(a%,b%,c%) As Variant
End Function

Sub Main
  a = Test(1,,4)          'El segundo parámetro fue omitido.
End Sub
```

Existen las siguientes restricciones:

1. La llamada no puede terminar con una coma. *a = Test(1,,)* sería un ejemplo no válido.
2. La llamada debe contener el número mínimo de parámetros requeridos por la función. Los siguientes ejemplos no son válidos: *a = Test(,1)* (sólo se pasa uno de los parámetros requeridos) y *a = Test(1,2)* (sólo se pasan dos de los parámetros requeridos).

Cuando se salta un parámetro de esta manera, BasicScript crea una variable temporal y pasa esta variable en su lugar. El valor de la variable que se pasa se corresponde con los mismos que retorna una función a la que no se ha asignado un valor.

En el siguiente ejemplo se muestra la declaración de la función *Factorial* (versión iterativa) y cómo se realiza una llamada:

```
Function Factorial(n%) As Integer
  f% = 1

  For i = n To 2 Step -1
    f = f * i
  Next i

  Factorial = f
End Function

Sub Main()
  a% = 0

  Do While a% < 2
    a% = Val(InputBox$("Teclee un entero mayor de 2.","Calcular el factorial"))
  Loop

  b# = Factorial(a%)
  MsgBox "El factorial de " & a% & " es: " & b#
End Sub
```

Salida forzada de las funciones

La sentencia *Exit Function* fuerza la salida de la función actual, continuando la ejecución por la sentencia que sigue a la llamada a la función. Sólo puede aparecer en el bloque de una función. En el siguiente ejemplo la función visualiza un mensaje y entonces termina con *Exit*:

```
Function Test_Exit() As Integer
  MsgBox "Probando la función Exit, volviendo a Main()."
  Test_Exit = 0
  Exit Function
  MsgBox "Esta línea nunca debería ser ejecutada."
End Function

Sub Main()
  a% = Test_Exit()
  MsgBox "Esta es la última línea de Main()."
End Sub
```

2.4.3 Declaración de subrutinas

La sintáxis para declarar una subrutina es la siguiente:

[Private | Public] [Static] Sub *Name*[(*Arglist*)]
  [*Statements*]
End Sub

El parámetro *Arglist* es una lista de hasta 30 argumentos (separados por comas) que siguen la sintáxis:

[Optional] [ByVal | ByRef] *Parameter*[()] [As *Type*]

La sentencia *Sub* tiene las siguientes partes:

Private Para indicar que la subrutina no puede llamarse desde otros *scripts*.

Public Todas las subrutinas son públicas por defecto: esto quiere decir que pueden ser llamadas desde otros *scripts*.

Static Sin efecto actualmente.

Name Nombre de la subrutina: sigue la convención habitual.

Optional Los parámetros que siguen serán opcionales: todos ellos deberán ser de tipo *Variant*. Se puede utilizar la función *IsMissing* para determinar si un parámetro opcional fue pasado o no en la llamada.

ByVal Para indicar que el parámetro se pasa por valor.

ByRef Para indicar que el parámetro se pasa por referencia (por defecto).

Parameter Nombre del parámetro: sigue la convención habitual.

Type Tipo del parámetro. Un array debe ser pasado con "()".

Una subrutina termina cuando se llega a una de las siguientes sentencias: *End Sub* y *Exit Sub*. Las subrutinas pueden ser recursivas.

Salida forzada de subrutinas

La sentencia *Exit Sub* fuerza la salida de la subrutina actual, continuando la ejecución por la sentencia que sigue a la llamada de la subrutina. Sólo puede aparecer dentro del bloque de una subrutina. En el siguiente ejemplo la subrutina visualiza una ventana de diálogo y entonces finaliza. La última línea no debería ser nunca ejecutada a causa de la sentencia *Exit Sub*.

```
Sub Main()
  MsgBox "Terminando Main()."
  Exit Sub
  MsgBox "Sigue todavía el Main()."
End Sub
```

2.4.4 Algunos aspectos del paso de parámetros

Dato fundamental a *Variant*

Al pasar el dato por referencia se previene que el *Variant* cambie su tipo dentro de la rutina. Por ejemplo:

```
Sub Foo(v As Variant)
  v = 50                'OK.
  v = "Hola, mundo"    'Aquí se produce un error de tipo.
End Sub

Sub Main()
  Dim i As Integer
  Foo i                 'Pasa un entero por referencia.
End Sub
```

Variant a dato fundamental

Un *Variant* no puede pasarse a una rutina que acepte datos fundamentales por referencia. Por ejemplo:

```
Sub Foo(i as Integer)
  ...
End Sub

Sub Main()
  Dim a As Variant
  Foo a            'Aquí el compilador da un error de tipo.
End Sub
```

Paso de estructuras

Las UDT's pueden pasarse tanto a rutinas definidas por el usuario como a rutinas externas. Siempre se pasan por referencia: la palabra clave *ByVal* no puede utilizarse cuando se definen argumentos de tipo estructura que se pasan a rutinas externas (mediante *Declare*). *ByVal* sólo se podrá utilizar con tipos de datos fundamentales. En realidad, cuando se pasa una estructura a una rutina externa sólo se pasa un puntero lejano a la estructura de datos.

Paso de arrays

Los arrays siempre se pasan por referencia. Por ejemplo:

```
Dim a(10) As String

FileList a          'Ambas llamadas están bien.
FileList a()
```

Parámetros nominales

Muchos elementos del lenguaje soportan los parámetros nominales, que permiten la especificación de parámetros en una función o subrutina mediante su nombre, en lugar del tradicional orden predeterminado. Algunos ejemplos de una llamada a *MsgBox*:

- **Nominal**:
  - MsgBox Prompt:= "Hola mundo."
  - MsgBox Title:="Título", Prompt:="Hola mundo"
  - MsgBox HelpFile:="BASIC.HLP",Prompt:="Hola mundo", HelpContext:=10
- **Posicional**:
  - MsgBox "Hola mundo"
  - MsgBox "Hola mundo",,"Título"
  - MsgBox "Hola mundo",,,"BASIC.HLP",10

De esta forma, el código es más fácil de leer y además se evita tener que conocer el orden de los parámetros, sobre todo si los parámetros son muchos y/o opcionales. Se deben seguir las siguientes reglas:

- Los parámetros nominales deben utilizar el nombre del parámetro especificado en la descripción de ese elemento del lenguaje, de lo contrario habrá un error de compilación.
- Todos los parámetros (tanto nominales como opcionales) se separan por comas.
- El parámetro nominal y su valor asociado se separan con ":=".
- Si un parámetro es nominal, todos los parámetros que le siguen también deben serlo.

Por ejemplo:

```
MsgBox "Hola mundo", Title:="Título"        'OK
MsgBox Prompt:="Hola mundo",,"Título"   'MAL
```

2.5 Sentencias selectivas

2.5.1 La sentencia *If...Then...Else*

Permite al ejecución condicionada de una sentencia o grupo de sentencias. La sintáxis es:

If $Condition$ Then
  $Statements$
[Else
  $ElseStatements$]

Una segunda sintáxis añade la clásula $ElseIf$:

If $Condition$ Then
  [$Statements$]
[ElseIf $ElseCondition$ Then
  [$ElseIfStatements$]]
[Else
  [$ElseStatements$]]
End If

Los parámetros son:

Condition Cualquier expresión que resuelva un valor lógico.

Statements Una o más sentencias separadas por ";". Este grupo se ejecuta cuando la condición es $True$.

ElseStatements Una o más sentencias separadas por ";". Este grupo se ejecuta cuando todas las condiciones anteriores son $False$.

ElseCondition Cualquier expresión que resuelva un valor lógico. Se evalúa si todas las condiciones previas son $False$. Pueden aparecer cualquier número de clásulas $ElseIf$.

ElseIfStatements Una o más sentencias separadas por ";". Este grupo se ejecuta cuando su condición es $True$.

2.5.2 La sentencia *Select Case*

Es la sentencia de selección múltiple, según el valor de una determinada expresión. Su sintáxis es la siguiente:

Select Case $TestExpression$
[Case $ExpressionList$
  [$StatementBlock$]]
[Case $ExpressionList$
  [$StatementBlock$]]
  ...
[Case Else
  [$StatementBlock$]]
End Select

Esta sentencia tiene las siguientes partes:

TestExpression Una expresión numérica o de cadena.

StatementBlock Un grupo de sentencias de BasicScript. Si *TestExpression* se iguala con alguna de las expresiones contenidas en *ExpressionList*, entonces se ejecutará el bloque de sentencias.

ExpressionList Una lista de expresiones (separadas por comas) que serán comparadas con *TestExpression*.

Se pueden utilizar rangos de expresiones múltiples dentro de una clásula *Case* simple. Por ejemplo:

```
Case 1 To 10,12,15, Is > 40
```

Sólo se ejecutará el bloque que iguale primero la expresión. Si no se encuentra ninguno, entonces se ejecutará el bloque *Case Else*.

El siguiente ejemplo utiliza la sentencia *Select Case* para visualizar el sistema operativo actual:

```
Sub Main()
  OpSystem% = Basic.OS
  Select Case OpSystem%
    Case 0,2
      s = "Microsoft Windows"
    Case 3 to 8, 12
      s = "UNIX"
    Case 10
      s = "IBM OS/2"
    Case Else
      s = "Otro sistema"
  End Select

  MsgBox "Esta versión de BasicScript está ejecutándose sobre: " & s
End Sub
```

2.5.3 Funciones selectivas

A continuación se muestran tres funciones de muy alto nivel que permiten escribir sentencias selectivas en una sóla línea de código:

Choose Recibe un índice y hasta 13 expresiones como parámetros y devuelve la expresión que se encuentra en la posición indicada.

IIf Recibe una expresión lógica y otras dos expresiones (partes cierta y falsa) y devolverá una u otra según la evaluación de la expresión lógica.

Switch Recibe hasta siete pares condición–expresión y devuelve la expresión que corresponde a la primer condición que sea *True*.

2.6 Sentencias iterativas

2.6.1 El bucle *Do...Loop*

Se pueden utilizar hasta tres tipos de bucles:

Do {While | Until} *Condition*
  Statements
Loop

Do
  Statements
Loop {While | Until} *Condition*

Do
  Statements
Loop

Como en cualquier otro lenguaje, el bloque se repite mientras la condición sea verdadera o hasta que la condición sea verdadera. El bucle es infinito cuando no se especifica la cláusula de condición. La condición siempre será cualquier expresión lógica.

A continuación se muestran varios ejemplos. El primero de ellos utiliza la sentencia *Do . . . while* que ejecuta el cuerpo de la iteración primero y entonces chequea la condición, repitiéndolo si la condición es cierta:

```
Dim a$(100)
i% = -1
Do
  i% = i% + 1
  If i% = 0 Then
    a(i%) = Dir$("*")
  Else
    a(i%) = Dir$
  End If
Loop While (a(i%) <> "" And i% <= 99)
r% = SelectBox(i% & " ficheros encontrados",,a)
```

El segundo ejemplo utiliza la sentencia *DoWhile . . . Loop*, que chequea primero la condición y ejecuta el cuerpo de la iteración si es cierta, volviendo de nuevo al punto de chequeo:

```
Dim a$(100)
i% = 0
a(i%) = Dir$("*")
Do While (a(i%) <> "" And i% <= 99)
  i% = i% + 1
  a(i%) = Dir$
Loop
r% = SelectBox(i% & " ficheros encontrados",,a)
```

El tercer ejemplo utiliza la sentencia *DoUntil . . . Loop*, que ejecuta la iteración hasta que la condición sea cierta, evaluando primero la condición:

```
Dim a$(100)
i% = 0
a(i%) = Dir$("*")
Do Until a(i%) = "" Or i% = 100
  i% = i% + 1
  a(i%) = Dir$
Loop
r% = SelectBox(i% & " ficheros encontrados",,a)
```

El cuarto ejemplo utiliza la sentencia *Do . . . Loop Until*, que ejecuta la iteración hasta que la condición sea cierta, evaluándola tras la ejecución del cuerpo:

```
Dim a$(100)
i% = -1
Do
  i% = i% + 1
  If i% = 0 Then
    a(i%) = Dir$("*")
  Else
    a(i%) = Dir$
  End If
Loop Until (a(i%) = "" Or i% = 100)
r% = SelectBox(i% & " ficheros encontrados",,a)
```

Salida forzada de los bucles (*Do...Loop*)

La sentencia *Exit Do* lleva la línea de ejecución a la sentencia que sigue a la cláusula *Loop*. Sólo podrá aparecer dentro de una sentencia *Do . . . Loop*. El siguiente ejemplo carga un array con las entradas de directorios al menos que haya más de 10 entradas (en este caso la sentencia *Do . . . Loop* termina la iteración):

```
Sub Main()
  Dim a$(5)
  Do
    i% = i% + 1

    If i% = 1 Then
      a(i%) = Dir$("*")
    Else
      a(i%) = Dir$
    End If

    If i% >= 10 Then Exit Do
  Loop While (a(i%) <> "")

  If i% = 10 Then
    MsgBox "¡" & i% & " entrada procesadas!"
  Else
    MsgBox "¡Menos de " & i% & " entradas procesadas!"
  End If
End Sub
```

2.6.2 El bucle *While...Wend*

Es una sentencia iterativa que repite la ejecución de una sentencia o grupo de ellas mientras una condición de entrada sea verdadera. La condición se chequea al inicio de cada iteración. La sintáxis es:

```
While Condition
  [Statements]
Wend
```

El siguiente ejemplo ejecuta el bucle hasta que el generador de números aleatorios devuelve el valor 1 (pero un máximo de 1000 veces, momento en el que se fuerza la salida):

```
Sub Main()
  x% = 0
  count% = 0

  While x% <> 1
    x% = Rnd(1)
    If count% > 1000 Then
      Exit Sub
    Else
      count% = count% + 1
    End If
  Wend

  MsgBox "El bucle se ejecutó " & count% & " veces."
End Sub
```

2.6.3 El bucle *For Each*

La sentencia *For Each* repite un bloque de sentencias para cada elemento de una colección o array. Su sintáxis es:

```
For Each Member In Group
  [Statements]
  [Exit For]
  [Statements]
Next [Member]
```

Esta sentencia tiene los siguientes parámetros:

Member Nombre de una variable para guardar un elemento en cada iteración del bucle.

Group Nombre de la colección o array.

Statements El bloque de sentencias.

La variable de la iteración es una copia del elemento de la colección o array, por lo que los cambios sobre estas variables no afectan a los elementos originales. Los elementos se recorren en el mismo orden en que están almacenados en memoria. La ejecución continúa hasta que no haya más elementos. Pueden anidarse los bucles *For Each* con otros bucles, por lo que *Next* afectará al más interno (*For Each* o *For Next*) en ese momento. El nombre de la variable para los miembros debe ser único.

El siguiente ejemplo muestra una subrutina que itera a través de los elementos de un array utilizando la sentencias *For ... Each*:

```
Sub Main()
  Dim a(3 To 10) As Single
  Dim i As Variant
  Dim s As String

  For i = 3 To 10
    a(i) = Rnd()
  Next i

  For Each i In a
    i = i + 1
  Next i

  s = ""
  For Each i In a
    If s <> "" Then s = s & ","
    s = s & i
  Next i

  MsgBox s
End Sub
```

En el siguiente ejemplo la subrutina visualiza los nombres de cada hoja de un libro de EXCEL:

```
Sub Main()
  Dim Excel As Object
  Dim Sheets As Object

  Set Excel = CreateObject("Excel.Application")

  Excel.Visible = 1
  Excel.Workbooks.Add

  Set Sheets = Excel.Worksheets

  For Each a In Sheets
    MsgBox a.Name
  Next a
End Sub
```

2.6.4 El bucle *For ... Next*

La sentencia *For ... Next* repite un bloque de sentencias un número especificado de veces. Su sintáxis es la siguiente:

```
For Counter = Start To End [Step Increment]
  [Statements]
  [Exit For]
  [Statements]
Next [Counter [,NextCounter] ... ]
```

Esta sentencia toma los siguientes parámetros:

Counter Nombre de una variable numérica (los tipos posibles son: *Integer*, *Long*, *Single*, *Double* o *Variant*).

Start Valor inicial del contador.

End Valor final del contador.

Increment Cantidad que se añade al contador en cada vuelta del bucle. Será positivo o negativo según los valores iniciales y finales. Por defecto, el incremento es +1. No se puede cambiar su valor durante la ejecución del bucle.

Statements El cuerpo de sentencias del bucle.

La sentencia continua su ejecución hasta que llega a una sentencia *ExitFor* o cuando el contador es mayor que el valor final (o menor si el bucle tiene recorrido inverso). Estos bucles pueden aparecer anidados. La sentencia *Next* se aplica al bucle *For...Next* más interno. La cláusula *Next* puede optimizarse agrupando varios contadores. Por ejemplo, los siguientes bucles son equivalentes:

```
For i = 1 To 10             For i = 1 To 10
  For j = 1 To 10              For j = 1 To 10
  Next j                    Next j,i
Next i
```

El siguiente ejemplo construye la tabla de verdad para la operación lógica *O* utilizando varios bucles *For...Next* anidados:

```
Sub Main()
  For x = -1 To 0
    For y = -1 To 0
      Z = x Or y
      msg = msg & Format(Abs(x%),"0") & " Or "
      msg = msg & Format(Abs(y%),"0") & " = "
      msg = msg & Format(Z,"True/False") & Basic.Eoln$
    Next y
  Next x
  MsgBox msg
End Sub
```

Salida forzada de los bucles (*For...Next*)

La sentencia *ExitFor* fuerza la salida del bucle más interno *For*, continuando la línea de ejecución tras la sentencia *Next*. Sólo puede aparecer en los bloques *For...Next*.

El siguiente ejemplo rellena un array con entradas de directorios hasta que se encuentra una entrada nula o se hayan procesado 100 entradas, momento en el que el bucle termina por una sentencia *Exit For*. La ventana de diálogo visualiza un contador de ficheros encontrados y algunas entradas del array:

```
Const crlf = Chr(13) + Chr(10)

Sub Main()
  Dim a$(100)
  For i = 1 To 100
    If i = 1 Then
      a$(i) = Dir$("*")
    Else
      a$(i) = Dir$
    End If

    If (a$(i) = "") Or (i >= 100) Then Exit For
  Next i

  msg = "Hay " & i & " ficheros encontrados." & crlf
  MsgBox msg & a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(10)
End Sub
```

Capítulo 3

Librería de extensión de tipos de RATIONAL ROSE

3.1 El objeto de la aplicación RATIONAL ROSE

Utilizando tanto ROSE *Script* como ROSE *Automation* se puede obtener el objeto de la aplicación RATIONAL ROSE.

En el primer caso, todos los *scripts* tienen por defecto un objeto global llamado *RoseApp*. Este objeto tiene la propiedad *CurrentModel* mediante la cual se puede abrir, controlar, salvar o cerrar un modelo de RATIONAL ROSE desde el *script*. Por ejemplo:

```
Sub GenerarCodigo (Modelo As Model)
  ' Aquí se genera algún código.
End Sub

Sub Main
  GenerarCodigo RoseApp.CurrentModel
End Sub
```

En el segundo caso, para utilizar RATIONAL ROSE como un servidor de automatización se debe inicializar previamente una instancia del objeto de la aplicación. Esto se realiza mediante una llamada a *CreateObject* o *GetObject* (o equivalentes) desde la aplicación que está actuando como controlador OLE. Estas llamadas devuelven el objeto OLE que implementa la API del objeto de la aplicación RATIONAL ROSE. Por ejemplo:

```
Sub GenerarCodigo (Modelo As Object)
  ' Aquí se genera algún código.
End Sub

Sub Main
  Dim RoseApp As Object
  Set RoseApp = CreateObject (“Rose.Application”)
  GenerarCodigo RoseApp.CurrentModel
End Sub
```

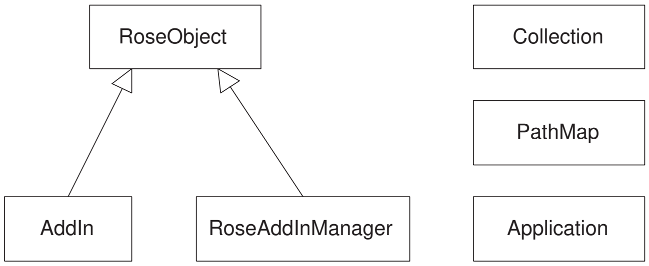

Figura 3.1: Diagrama de herencia de la clase *Application*.

3.2 Diagramas del modelo de herencia del Intefaz de Extensión de Rose REI

3.2.1 Diagrama de herencia de la clase *Application*

El diagrama de clases de la figura 3.1 muestra la estructura de herencia de la clase *Application* en el modelo REI. La clase ***RoseObject*** es la clase raíz de la jerarquía del modelo REI y ofrece los métodos apropiados que permiten saber cuál es el tipo de un objeto determinado (que representa algún elemento).

La clase *RoseObject* tiene los siguientes métodos:

CanTypeCast Comprobar si un objeto puede convertirse a un tipo determinado (sólo está disponible en los *scripts*).

GetObject Recuperar el objeto de automatización OLE del objeto.

IdentifyClass Identificar la clase del objeto.

IsClass Comprobar si el objeto es una instancia de una clase específica.

TypeCast Reconvertir el objeto a su tipo original (sólo está disponible en los *scripts*).

La clase ***RoseAddInManager*** tiene una única propiedad (*AddIns*) que contiene la colección de *Add–ins* activos en la aplicación. Las propiedades y métodos de la clase ***AddIn*** describen y controlan las características de cada *Add–ins* activo en la aplicación.

La mayoría de los elementos de un modelo de Rational Rose tienen su correspondiente colección: cada clase tiene una colección de la clase, cada categoría tiene una colección de la categoría, cada propiedad tienen una colección de la propiedad, etc. Estas colecciones son objetos de la clase ***Collection***.

Se utiliza la clase ***PathMap*** para crear y editar entradas del mapa de caminos para un modelo concreto.

La clase ***Application*** se utiliza para realizar las tareas habituales de Rational Rose. Las propiedades de esta clase son:

AddInManager La referencia a este objeto.

ApplicationPath El camino completo donde se encuentra el ejecutable activo (sólo lectura).

CommandLine La cadena que es pasada a la aplicación cuando se ejecuta.

CurrentModel El nombre del modelo actualmente abierto.

Height La altura de la ventana principal.

Left La posición del lado izquierdo de la ventana principal.

PathMap La referencia a este objeto.

ProductName El nombre de la aplicación activa.

Top La posición del lado superior de la ventana principal.

Version La versión actual de la aplicación activa.

Visible El control para indicar si la aplicación estará visible en la pantalla.

Width La anchura de la ventana principal.

Y los métodos de la clase *Application* son:

CompileScriptFile Compilar un fichero de *script*.

ExecuteScript Ejecutar el fuente o la imagen compilada de un *script*.

Exit Finalizar la ejecución de la aplicación.

FreeScript Descargar el *script* que está en memoria.

GetLicensedApplication Obtener una instacia de la aplicación dado su número de licencia.

GetObject Obtener el objeto de interfaz OLE asociado con la aplicación.

GetProfileString Obtener la cadena asociada con una entrada en el fichero *Rose.ini*.

LoadScript Cargar un *script* en la memoria para que pueda ser llamado por otros *scripts*.

NewModel Crear un nuevo modelo.

NewScript Abrir una ventana de edición en blanco para un nuevo *script*.

OpenExternalDocument Abrir un documento externo dado su nombre de fichero.

OpenModel Abrir un modelo existente.

OpenModelAsTemplate Abrir un modelo existente para utilizarlo como plantilla para crear otro nuevo.

OpenScript Abrir un *script* en la ventana de edición.

OpenURL Abrir un documento de Internet dada su URL.

Save Salvar el modelo actual.

SaveAs Salvar el modelo actual con un nuevo nombre.

WriteErrorLog Escribir un mensaje en la ventana del registro de acciones.

WriteProfileString Escribir una entrada y su cadena asociada en el fichero *Rose.ini*.

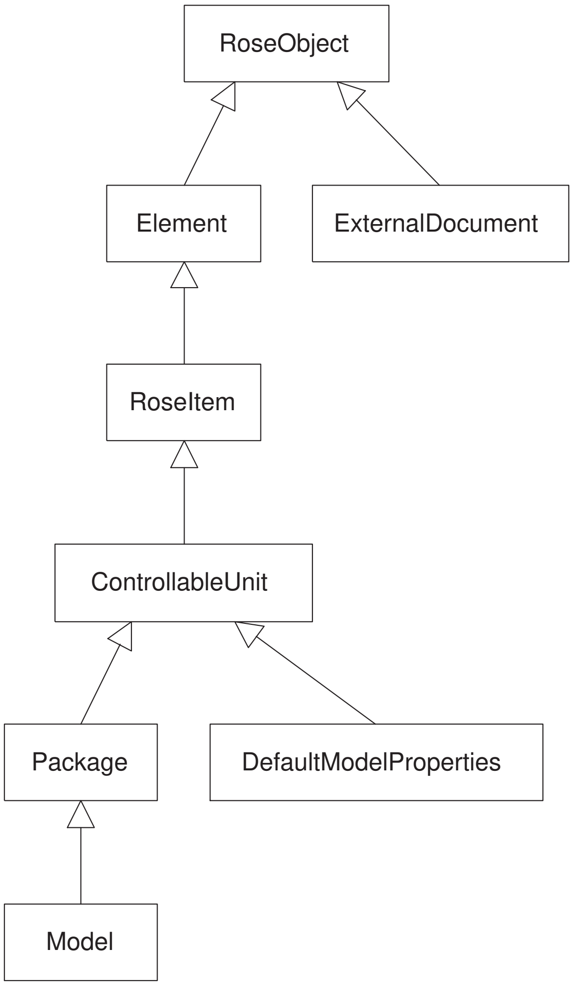

Figura 3.2: Diagrama de herencia de la clase *Model*.

3.2.2 Diagrama de herencia de la clase *Model*

El diagrama de clases de la figura 3.2 muestra la estructura de herencia de la clase *Model* en el modelo REI. La clase ***RoseObject*** es la clase base desde la cual derivan directa o indirectamente el resto de clases del modelo.

La clase ***Element*** ofrece el interfaz a las propiedades de cada elemento del modelo: cada objeto en el modelo (incluso el objeto *Model*) es un elemento. Cada elemento de un modelo tiene un nombre y un identificador (*ID*) único que será el medio más directo para acceder a un ítem de una colección. Las propiedades específicas de *Element* son:

Name El nombre del elemento.

Application La aplicación a la que pertenece el elemento.

Model El modelo al que pertenece el elemento.

Y los métodos específicos de *Element* son:

CreateProperty Crear una nueva propiedad para el elemento.

FindDefaultProperty Obtener la propiedad por defecto del elemento.

FindProperty Obtener una propiedad del elemento.

GetAllProperties Obtener la colección de propiedades del elemento.

GetCurrentPropertySetName Obtener el conjunto de propiedades actual del elemento.

GetDefaultPropertyValue Obtener el valor por defecto de una propiedad del elemento.

GetDefaultSetNames Obtener los nombres de los conjuntos de propiedades por defecto que pertenecen al elemento.

GetPropertyClassName Obtener el nombre de la clase que es requerido por el objeto *Model* cuando se trabaja con conjuntos de propiedades por defecto.

GetPropertyValue Obtener el valor actual de una propiedad del elemento.

GetQualifiedName Obtener el nombre calificado del elemento.

GetToolNames Obtener los nombres de las herramientas asociadas al elemento.

GetToolProperties Obtener la colección de propiedades para una herramienta asociada con el elemento.

GetUniqueId Obtener el *ID* del elemento.

InheritProperty Borrar el valor sobreescrito de una propiedad para utilizar su valor por defecto.

IsDefaultProperty Indicar si el valor actual de una propiedad es su valor por defecto.

IsOverriddenProperty Indicar si el valor actual de una propiedad ha sido sobreescrito.

OverrideProperty Sobreescribir el valor por defecto de una propiedad.

SetCurrentPropertySetName Especificar un conjunto de propiedades como el conjunto de propiedades actual del elemento.

Cada ***RoseItem*** es un elemento del modelo y por lo tanto hereda todas las propiedades y métodos de *Element*. Sus propiedades y métodos se utilizan para editar la especificación de cada ítem. Las propiedades específicas de *RoseItem* son:

Documentation La documentación que pertenece al ítem.

Stereotype El estereotipo del ítem.

ExternalDocuments La colección de documentos externos asociados al ítem.

LocalizedStereotype El equivalente localizado a la propiedad *stereotype*.

Y los métodos específicos de *RoseItem* son:

AddExternalDocument Asociar un documento externo al ítem.

DeleteExternalDocument Borrar un documento externo asociado al ítem.

GetRoseItem Recuperar el ítem como un objeto OLE.

OpenSpecification Abre la ventana de especificación del ítem.

La clase ***ControllableUnit*** es una clase abstracta que ofrece la funcionalidad de una unidad controlable de RATIONAL ROSE en el interfaz de extensión REI. Los métodos específicos de *ControllableUnit* son:

Control Asociar una unidad controlable con un fichero que se ha pasado a una aplicación de gestión de configuración.

GetFileName Obtener el nombre del fichero que contiene la unidad controlable.

IsControlled Indicar si la unidad controlable está controlada.

IsLoaded Indicar si una unidad controlable está cargada en el modelo actual.

IsModifiable Indicar si la unidad controlable puede ser modificada.

IsModified Indicar si la unidad controlable ha sido modificada.

Load Cargar una unidad controlable en el modelo actual.

Modifiable Permitir que una unidad controlable sea modificada.

SaveAs Salvar una unidad controlable con un nuevo nombre.

Save Salvar una unidad controlable.

Unload Descargar una unidad controlable del modelo actual.

Uncontrol Finalizar el control sobre una unidad controlable.

La clase ***Package*** es un contenedor para los elementos del modelo que corresponden al concepto de *Paquete de* UML. Los métodos de la clase *Package* permiten determinar si es el paquete raíz del modelo (*IsRootPackage*) y obtener el objeto OLE asociado con el paquete.

Una vez que se utilizan los métodos de la clase *Application* para establecer el modelo actual, la clase ***Model*** facilita las propiedades y métodos que permiten trabajar con los objetos del modelo. Por ejemplo, se pueden añadir y obtener los objetos del modelo (clases, categorías, relaciones, procesadores, dispositivos, diagramas, etc.), borrar objetos del modelo, etc. Las propiedades específicas de la clase *Model* son:

DefaultProperties Las propiedades por defecto del modelo.

DeploymentDiagram El diagrama de despliegue asociado al modelo.

RootCategory La categoría *Top Level* de RATIONAL ROSE. Se corresponde a la vista lógica del modelo. Sólo lectura.

RootSubsystem El subsistema *Top Level* de RATIONAL ROSE. Se corresponde a la vista de componentes del modelo. Sólo lectura.

RootUseCaseCategory La categoría raíz a la que pertenecen los casos de usos. Se corresponde a la vista de casos de uso del modelo. Sólo lectura.

UseCases Los casos de usos que pertenecen al modelo.

Y los métodos específicos de la clase *Model* son:

FindCategories Encontrar una colección de categorías del modelo.

FindCategoryWithID Encontrar una categoría específica del modelo.

FindClassWithID Encontrar una clase específica del modelo.

FindClassses Encontrar una colección de clases del modelo.

FindItemWithID Encontrar un ítem específico del modelo.

FindItems Encontrar una colección de ítemes del modelo.

GetAllClasses Obtener todas las clases de todas las categorías del modelo.

GetAllCategories Obtener todas las categorías del modelo.

GetAllSubsystems Obtener todos los subsistemas del modelo.

GetAllModules Obtener todos los módulos del modelo.

GetAllProcessors Obtener todos los procesadores del modelo.

GetAllDevices Obtener todos los dispositivos del modelo.

GetAllUseCases Obtener todos los casos de uso del modelo.

GetActiveDiagram Obtener el diagrama actualmente activo del modelo.

GetSelectedClasses Obtener todas las clases actualmente seleccionadas en el modelo.

GetSelectedCategories Obtener todas las categorías actualmente seleccionadas en el modelo.

GetSelectedModules Obtener todos los módulos actualmente seleccionados en el modelo.

GetSelectedSubsystems Obtener todos los subsistemas actualmente seleccionados en el modelo.

GetSelectedUsesCases Obtener todos los casos de uso actualmente seleccionados en el modelo.

AddProcessor Añadir un procesador al modelo.

DeleteProcessor Eliminar un procesador del modelo.

AddDevice Añadir un dispositivo al modelo.

DeleteDevice Eliminar un dispositivo del modelo.

La clase ***DefaultModelProperties*** es un contenedor para las propiedades por defecto que pertenecen al modelo. Sólo hay un objeto *DefaultModelProperties* por modelo. Los métodos específicos de esta clase son:

AddDefaultProperty Añadir una propiedad por defecto a un conjunto de propiedades.

CloneDefaultPropertySet Duplicar un conjunto de propiedades por defecto para utilizarlo como base para crear un nuevo conjunto de propiedades.

CreateDefaultPropertySet Crear un nuevo conjunto de propiedades por defecto (en blanco).

DeleteDefaultProperty Eliminar una propiedad por defecto del conjunto de propiedades por defecto.

DeleteDefaultPropertySet Eliminar una propiedad por defecto de un modelo.

FindDefaultProperty Encontrar una propiedad por defecto especificada dados un modelo, una clase y el nombre de una herramienta.

GetDefaultPropertySet Obtener el conjunto de propiedades por defecto para una herramienta y clase dadas.

GetDefaultSetNames Obtener los nombres de los conjuntos de propiedades por defecto para una herramienta y clase dadas.

GetToolNamesForClass Obtener los nombres de las herramientas asociadas a una clase dada.

IsToolVisible Determinar si la página con las propiedades por defecto del modelo de una herramienta dada está visible en las especificaciones.

SetToolVisibility Establecer la visibilidad de una herramienta, es decir, si la página de propiedades para esa herramienta aparecerá en las especificaciones.

Todas las relaciones (*ClassRelation*, *Inherits*, *Has* y *Realizes*) heredan de la clase ***Relation***: sus propiedades y métodos permiten especificar y obtener la información de todas las partes involucradas en cada relación del modelo. Véase la subsección sobre el diagrama de herencias de *Relation*.

La clase ***ExternalDocument*** ofrece las propiedades y métodos que permiten crear documentos externos o informes desde el entorno de RATIONAL ROSE. Por ejemplo, se puede ejecutar WORD en WINDOWS para pasar la información de un modelo a un documento de WORD. Las propiedades específicas de *ExternalDocument* son:

ParentCategory La aplicación a utilizar para abrir el documento.

Path El camino completo al documento.

URL La URL del documento en Internet. Las propiedades *Path* y *URL* son excluyentes entre sí.

Y los métodos específicos de *ExternalDocument* son:

IsURL Chequear si el documento tiene una URL.

Open Abrir un documento externo.

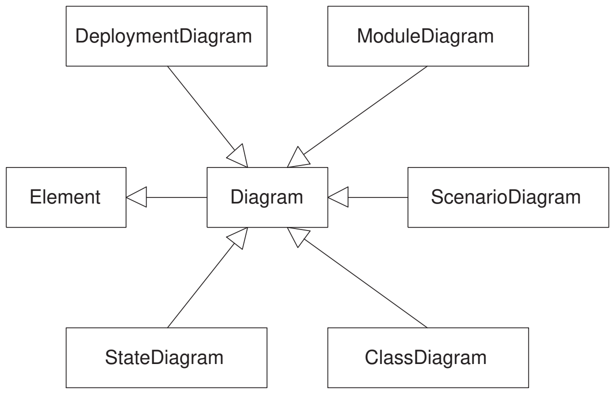

Figura 3.3: Diagrama de herencia de la clase *Diagram*.

3.2.3 Diagrama de herencia de la clase *Diagram*

El diagrama de clases de la figura 3.3 muestra la estructura de herencia de la clase *Diagram* en el modelo REI.

Ya se ha comentado que cada objeto en un modelo es un elemento, así que la clase ***Diagram*** está directamente subordinada a *Element*. La clase *Diagram* ofrece un conjunto de propiedades y métodos que todas las demás clases de diagramas (por ejemplo, diagrama de clases, diagrama de escenarios, diagrama de componentes, etc.) heredan. Cualquier diagrama se compone de varios elementos (*items*) y vistas de esos elementos (*ItemViews*). Cada *Itemview* es la representación física de un elemento de Rational Rose y es un objeto con las propiedades y métodos que definen su apariencia en la ventana del diagrama (posición, color, tamaño, etc.). La operatividad básica de la clase *Diagram* se resume en los siguientes puntos:

- La propiedad *Diagram.ItemViews* es utilizada para iterar a través de la colección de vistas de elementos que pertenecen al diagrama.
- La propiedad *Diagram.Items* es utilizada para iterar a través de los elementos que existen en el diagrama.
- El método *Diagram.GetViewFrom* es utilizado para encontrar la primera vista de un elemento.
- Para encontrar cuáles son las vistas de los elementos actualmente seleccionados en el diagrama, se recorre la lista de vistas de elementos del diagrama. Para cada una de ellas, se utiliza el método *ItemView.IsSelected* para determinar si está o no seleccionado en el diagrama. Se puede obtener entonces la vista de elemento seleccionada o realizar cualquier otro proceso basándose en esta información.
- Una manera rápida para obtener todos los elementos seleccionados en un diagrama es utilizar el método *Diagram.GetSelectedItems*. En lugar de iterar a través del diagrama y chequear cada vista, este método simplemente devuelve todo aquello que esté seleccionado.

Otras propiedades específicas de *Diagram* son:

Documentation La documentación anexa al diagrama.

Visible La visibilidad del diagrama.

Otros métodos específicos de *Diagram* son:

Exists Comprobar si un determinado objeto del diagrama existe.

Layout Dibujar un diagrama.

Invalidate Redibujar un diagrama.

Update Modificar un diagrama.

AddNoteView Añadir una nota a un diagrama.

RemoveNoteView Borrar una nota del diagrama.

GetNoteViews Obtener las notas pertenecientes al diagrama.

Render Salvar el gráfico del diagrama en un fichero con el formato *Windows Metafile*.

RenderEnhanced La misma acción que *Render*, pero con mayor calidad.

RenderToClipboard La misma acción que *Render*, pero guarda el gráfico en el portapapeles.

RenderEnhancedToClipBoard La misma acción que *RenderToClipboard*, pero con mayor calidad.

IsActive Indicar si el diagrama está actualmente activo en la aplicación.

Activate Activar el diagrama.

La clase ***ClassDiagram*** permite añadir, obtener y borrar clases y categorías del diagrama de clases. Las propiedades específicas de *ClassDiagram* son:

ParentCategory La categoría que contiene al diagrama de clases.

Y los métodos específicos de *ClassDiagram* son:

AddAssociation Añadir una asociación al diagrama.

AddCategory Añadir una categoría al diagrama.

AddClass Añadir una clase al diagrama.

AddUseCase Añadir un caso de uso al diagrama.

GetAssociations Obtener la colección de las asociaciones que pertenecen al diagrama de clases.

GetCategories Obtener la colección de las categorías que pertenecen al diagrama de clases.

GetClasses Obtener la colección de las clases que pertenecen al diagrama de clases.

GetClassView Obtener una vista de clase de un diagrama de clases.

GetSelectedCategories Obtener las categorías actualmente seleccionadas en el diagrama de clases.

GetSelectedClasses Obtener las clases actualmente seleccionadas en el diagrama de clases.

GetUseCases Obtener los casos de uso pertenecientes al diagrama.

IsUseCaseDiagram Determinar si el diagrama de clases es un diagrama de casos de uso.

RemoveAssociation Eliminar una asociación del diagrama de clases.

RemoveCategory Eliminar una categoría del diagrama de clases.

RemoveClass Eliminar una clase del diagrama de clases.

RemoveUseCase Eliminar un caso de uso del diagrama.

Un escenario es una instancia de un caso de uso y muestra los eventos que ocurren durante la ejecución del sistema. La clase ***ScenarioDiagram*** permite crear una representación visual de un escenario. Las propiedades específicas de *ScenarioDiagram* son:

InstanceViews La colección de vistas de instancias pertenecientes al diagrama.

Y los métodos específicos de *ScenarioDiagram* son:

AddInstance Añadir una instancia al diagrama de escenario.

AddInstanceView Añadir una vista de instancia al diagrama de escenario.

CreateMessage Crear un mensaje y añadirlo al diagrama de escenario.

DeleteInstance Eliminar una instancia del diagrama de escenario.

GetDiagramType Obtener el valor del tipo de diagrama.

GetMessages Obtener la colección de mensajes que pertenecen al diagrama de escenario.

GetSelectedLinks Obtener los enlaces actualmente seleccionados en el diagrama de escanario.

GetSelectedMessages Obtener el conjunto de mensajes actualmente seleccionados en el diagrama de escenario.

GetSelectedObjects Obtener el conjunto de objetos actualmente seleccionados en el diagrama de escanario.

RemoveInstanceView Eliminar una vista de instancia del diagrama de escenario.

La clase ***StateDiagram*** define las propiedades y métodos que permiten la manipulación de objetos en la vista de estados (contiene los diagramas de estado) de un modelo. Las propiedades específicas de la clase *StateDiagram* son:

Parent La máquina de estados que contiene el diagrama de estados.

Y los métodos específicos de *StateDiagram* son:

AddStateView Añadir una vista de estados a un diagrama de estados.

GetSelectedStates Obtener los estados actualmente seleccionados en un diagrama de estados.

GetSelectedStateViews Obtener las vistas de estados actualmente seleccionadas en un diagrama de estados.

GetSelectedTransitions Obtener las transiciones actualmente seleccionadas en un diagrama de estados.

GetStateView Obtener el estado de una vista de estados en un diagrama de estados.

GetStateViews Obtener todas las vistas de estados en un diagrama de estados.

RemoveStateView Eliminar una vista de estados de un diagrama de estados.

Un ***ModuleDiagram*** mapea la colocación de las clases y objetos en módulos. Las propiedades específicas de *ModuleDiagram* son:

ParentSubsystem El subsistema que contiene el módulo. Siempre hace referencia a un objeto *Subsystem* válido.

Y los métodos específicos de *ModuleDiagram* son:

AddModule Añadir un módulo al diagrama de módulos.

AddSubsystem Añadir un subsistema al diagrama de módulos.

GetModules Obtener la colección que contiene todos los módulos pertenecientes al diagrama de módulos.

GetSelectedModules Obtener los módulos actualmente seleccionados en el diagrama de módulos.

GetSelectedSubsystems Obtener los susbsistemas actualmente seleccionados en el diagrama de módulos.

GetSubsystems Obtener la colección que contiene todos los subsistemas que pertenecen al diagrama de módulo.

Un diagrama de despliegue es la representación visual de los dispositivos y procesadores físicos. La clase ***DeploymentDiagram*** ofrece las siguientes métodos específicos:

AddDevice Añadir un dispositivo al diagrama de despliegue.

AddProcessor Añadir un procesador al diagrama de despliegue.

GetDevices Obtener los dispositivos que pertenecen al diagrama de despliegue.

GetProcessors Obtener los procesadores que pertenecen al diagrama de despliegue.

RemoveDevice Eliminar un dispositivo del diagrama de despliegue.

RemoveProcessor Eliminar un procesador del diagrama de despliegue.

3.2.4 Diagrama de herencia de la clase *Logical*

El diagrama de clases de la figura 3.4 muestra la estructura de herencia de las dos principales clases lógicas del modelo REI: *Category* y *Class*. Ambas heredan de las clases *Package* y *RoseItem* respectivamente, que pertenecen al diagrama de herencia de la clase *Model*.

La clase ***Category*** permite definir y manipular colecciones lógicas de clases. Las propiedades especificas de la clase *Category* son:

Associations La colección de todas las asociaciones que pertenecen a la categoría.

Categories La colección de las categorías que son hijas de las categoría.

ClassDiagrams La colección de los diagramas de clase que pertenecen a la categoría.

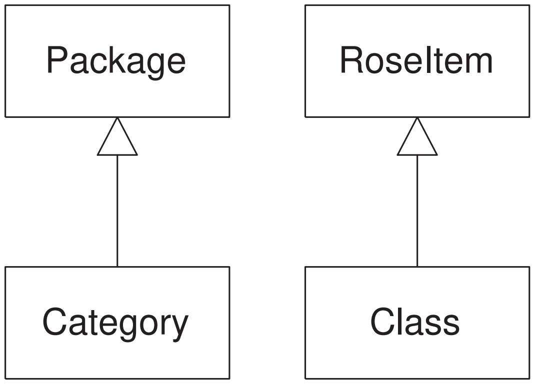

Figura 3.4: Diagrama de herencia de la clase *Logical*.

Classes La colección de las clases que pertenecen a la categoría.

Global Identifica la categoría como global.

ParentCategory La categoría que contiene a la categoría.

ScenarioDiagrams La colección de los diagramas de escenario que pertenecen a la categoría.

UseCases La colección que contiene los casos de uso que pertenecen a la categoría.

Y los métodos específicos de la clase *Category* son:

AddCategory Añadir una categoría a una categoría.

AddCategoryDependency Añadir una dependencia de categoría a una categoría.

AddClass Añadir una clase a una categoría.

AddClassDiagram Añadir un diagrama de clase a una categoría.

AddScenarioDiagram Añadir un diagrama de escenario a una categoría.

AddUseCase Añadir un caso de uso a una categoría.

DeleteCategory Eliminar una categoría de una categoría.

DeleteCategoryDependency Eliminar una dependencia de categoría de una categoría.

DeleteClass Eliminar una clase de una categoría.

DeleteClassDiagram Eliminar un diagrama de clase de una categoría.

DeleteScenarioDiagram Eliminar un diagrama de escenario de una categoría.

DeleteUseCase Eliminar un caso de uso de una categoría.

GetAllCategories Obtener todas las categorías que pertenecen a una categoría y todas las categorías pertenecientes a sus hijas.

GetAllClasses Obtener todas las clases que pertenecen a la categoría y todas sus hijas.

GetAllUseCases Obtener la colección que contiene los casos de uso que pertenecen a la categoría.

GetAssignedSubsystem Obtener el subsistema asignado a la categoría.

GetCategoryDependencies Obtener la colección de dependencias de categoría que pertenecen a la categoría.

HasAssignedSubsystem Obtener el subsistema asignado a la categoría.

RelocateCategory Recolocar una categoría dentro de una categoría.

RelocateClass Recolocar una clase dentro de una categoría.

RelocateClassDiagram Recolocar un diagrama de clase dentro de una categoría.

RelocateScenarioDiagram Recolocar un diagrama de escenario dentro de una categoría.

SetAssignedSubsystem Asignar un subsistema a la categoría.

TopLevel Indicar si la categoría es la categoría raíz.

La clase ***Class*** permite establecer y obtener las características y relaciones de las clases específicas de un modelo: si es una clase abstracta, si es un tipo fundamental, si es persistente, si puede ser concurrente con otras clases, su conjunto de atributos y operaciones, sus relaciones con otros objetos del modelo, etc. Las propiedades específicas de la clase *Class* son:

ParentCategory La categoría que contiene la clase.

Abstract Identifica la clase como abstracta.

Attributes La colección que contiene los atributos de la clase.

Operations La colección que contiene las operaciones de la clase.

ExportControl La visibilidad de la clase.

Cardinality La cardinalidad de la clase.

ClassKind El tipo de la clase.

Concurrency El tipo de concurrencia de la clase con otros objetos.

Persistence Identifica la clase como persistente.

Space El algoritmo de espacios a utilizar para la clase.

FundamentalType Identifica la clase como un tipo fundamental.

StateMachine La máquina de estado que pertenece a la clase.

ParentClass La clase padre de esta clase.

Y los métodos específicos de la clase *Class* son:

AddAssignedModule Asignar un módulo (componente) a la clase.

AddAssociation Añadir una asociación (*Association*) a una clase.

AddAttribute Añadir un atributo a una clase.

AddClassDependency Añadir una dependencia de clase a la clase.

AddHas Añadir una agregación (*HasRelationship*) a una clase.

AddInheritRel Añadir una relación de herencia (*InheritRelationship*) entre clases en una clase.

AddNestedClass Añadir una clase anidada a una clase.

AddOperation Añadir una operación a una clase.

AddRealizeRel Añadir una relación de realización a una clase.

CreateStateMachine Crear una máquina de estado para una clase.

DeleteAssociation Eliminar una asociación de una clase.

DeleteAttribute Eliminar un atributo de una clase.

DeleteClassDependency Eliminar una dependencia de clase de una clase.

DeleteHas Eliminar una agregación de una clase.

DeleteInheritRel Eliminar una relación de herencia entre clases en una clase.

DeleteNestedClass Eliminar una clase anidada de una clase.

DeleteOperation Eliminar una operación de una clase.

DeleteRealizeRel Eliminar una relación de realización de una clase.

DeleteStateMachine Eliminar la máquina de estados de la clase del modelo.

GetAssignedLanguage Obtener el nombre del lenguaje de programación asignado a la clase.

GetAssignedModules Obtener los módulos asignados a la clase.

GetAssociateRoles Obtener los roles asociados a la clase.

GetAssociations Obtener las asociaciones en las que participa la clase.

GetClassDependencies Obtener la colección de clases de dependencia que pertenecen a la clase.

GetHasRelations Obtener las agregaciones en las que participa la clase.

GetInheritRelations Obtener las relaciones de herencia en las que participa la clase.

GetLinkAssociation Obtener la asociación para una clase de enlace (*Link*).

GetNestedClasses Obtener las clases anidadas en la clase.

GetRealizeRelations Obtener la colección de relaciones de realización que pertenecen a la clase.

GetRoles Obtener los roles de la clase.

GetSubClasses Obttener las subclases de la clase.

GetSuperclasses Obtener las superclases correspondientes a la clase.

IsALinkClass Determinar si una clase es una clase de enlace.

IsNestedClass Determinar si una clase es una clase anidada.

RemoveAssignedModule Borrar una asignación de módulo de la clase.

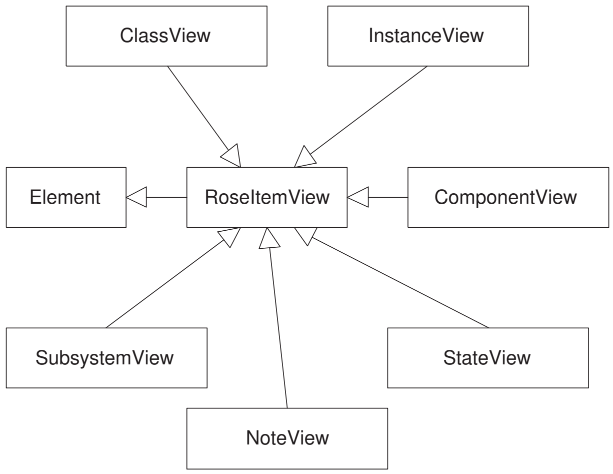

Figura 3.5: Diagrama de herencia de la clase *View*.

3.2.5 Diagrama de herencia de la clase *View*

El diagrama de clases de la figura 3.5 muestra la estructura de herencia de la clase $View$ en el modelo REI. La clase $RoseItemView$ hereda de $Element$, que pertenece al diagrama de herencia de la clase $Model$.

La clase ***RoseItemView*** permite definir y manipular el tamaño y posición de un $RoseItem$ en un diagrama. Las propiedades especificas de la clase $RoseItemView$ son:

Name El nombre de la vista del ítem.

Item El objeto $RoseItem$ representado por la vista.

ParentDiagram El diagrama que contiene a la vista.

ParentView El objeto $RoseItemView$ que contiene a la vista.

SubViews La colección de vistas de ítem que pertenecen a la vista.

XPosition La coordenada horizontal (x) del punto central de la vista.

YPosition La coordenada vertical (x) del punto central de la vista.

Height La altura de la vista.

Width La anchura de la vista.

FillColor.Red Establece el color de la vista a rojo.

FillColor.Green Establece el color de la vista a verde.

FillColor.Blue Establece el color de la vista a azul.

FillColor.Transparent Establece el color de la vista a transparente.

LineColor.Red Establece el color de línea de la vista a rojo.

LineColor.Green Establece el color de línea de la vista a verde.

LineColor.Blue Establece el color de línea de la vista a azul.

Font.Red Establece el color del texto a rojo.

Font.Green Establece el color del texto a verde.

Font.Blue Establece el color del texto a azul.

Font.FaceName Establece el tipo de letra fuente del texto (Arial, Courier, etc.).

Font.Size Establece el tamaño (en puntos) de la fuente del texto.

Font.Bold Indica si el estilo de la fuente del texto es *Bold* ().

Font.Italic Indica si el estilo de la fuente del texto es *Italic* (itálica).

Font.Underline Indica si el estilo de la fuente del texto es *Underline* (subrayado).

Font.StrikeThrough Indica si el estilo de la fuente del texto es *StrikeThrough* (tachado).

Y los métodos específicos de *RoseItemView* son:

Invalidate Redibujar la vista especificada en la pantalla.

GetDefaultHeight Obtener la altura por defecto de la vista (calculada por ROSE).

GetDefaultWidth Obtener la anchura por defecto de la vista (calculada por ROSE).

GetMinHeight Obtener la altura mínima de la vista (calculada por ROSE).

GetMinWidth Obtener la anchura mínima de la vista (calculada por ROSE).

HasItem Determinar si la vista tiene su objeto *RoseItem* correspondiente.

HasParentView Determinar si la vista pertenece a otra vista.

IsSelected Determinar si la vista está actualmente seleccionada en el diagrama.

PointInView Determinar si una coordenada dada (x,y) está dentro de la vista especificada.

SetSelected Seleccionar la vista en el diagrama.

SupportsFillColor Permitir a la vista utilizar un color de relleno (si es apropiado).

SupportsLineColor Permitir a la vista utilizar un color de línea (si es apropiado).

La clase ***ClassView*** es la representación visual de una clase y establece su apariencia en un diagrama del modelo. La clase hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista de la clase en un diagrama. Las propiedades específicas de *ClassView* son:

AutomaticResize Si el redimensionado automático de la clase está activado.

ShowOperationsSignature Si las signaturas de las operaciones de la clase son mostradas.

ShowAllAttributes Si los atributos de la clase son visibles.

ShowAllOperations Si las operaciones de la clase son visibles.

SuppressAttributes Si los atributos de la clase son suprimidos.

SuppressOperations Si las operaciones de la clase son suprimidas.

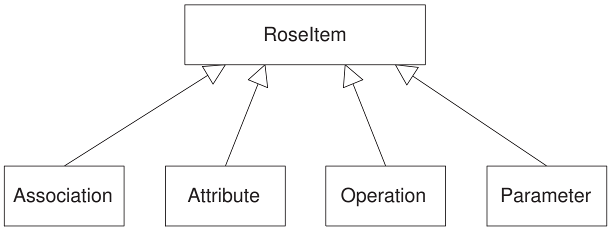

Figura 3.6: Diagrama de herencia de la clase *Related Logical*.

La clase ***InstanceView*** hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista de la clase en un diagrama. Sólo tiene un método específico: *GetInstance* (obtener el objeto *Instance* representado por la vista).

La clase ***ComponentView*** es la representación visual de un componente y establece su apariencia en un diagrama del modelo. La clase hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista del componente en un diagrama. Sólo tiene un método específico: *GetComponent* (obtener el objeto *Component* representado por la vista).

La clase ***StateView*** hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista de la clase en un diagrama. Sólo tiene un método específico: *GetState* (obtener el estado actual de la vista).

La clase ***NoteView*** hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista de la clase en un diagrama. Tiene una propiedad y un método específicos: *Text* (el texto que aparece en la vista de la nota) y *GetNoteViewType* (obtener el tipo de la vista de la nota, es decir, si la nota aparece en una caja o es flotante).

La clase ***SubsystemView*** es la representación visual de un subsistema y establece su apariencia en un diagrama del modelo. La clase hereda de *RoseItemView* las propiedades y métodos para determinar la posición y tamaño de una vista del componente en un diagrama. Sólo tiene un método específico: *GetSubsystem* (obtener el objeto *Subsystem* representado por la vista).

3.2.6 Diagrama de herencia de la clase *Related Logical*

El diagrama de clases de la figura 3.6 muestra la estructura de herencia de la clase *RelatedLogical* en el modelo REI. Todas las clases heredan de *RoseItem* que pertenece al diagrama de herencia de la clase *Model*.

Una asociación es una conexión o enlace entre clases. La clase ***Association*** exponen un conjunto de propiedades y métodos que permiten determinar las características de las asociaciones entre las clases y obtener las asociaciones existentes en el modelo. Las propiedades específicas de *Association* son:

Derived Identifica la asociación como derivada.

LinkClass Identifica la asociación como una clase de enlace.

Role1 El primer rol de la asociación.

Role2 El segundo rol de la asociación.

Roles La colección de roles que pertenecen a la asociación.

Y los métodos específicos de *Association* son:

ClearRoleForNameDirection Elimina la dirección del nombre de un rol que pertenece a la asocación.

GetCorrespondingRole Obtener el rol correspondiente a una asociación

GetOtherRole Obtener otro rol de una asociación.

GetRoleForNameDirection Obtener el rol que es la dirección del nombre de la asociación.

NameIsDirectional Determinar si la asociación tiene una dirección del nombre.

SetLinkClassName Especificar la clase que es el enlace de clase para la asociación.

SetRoleForNameDirection Especificar un rol como la dirección del nombre de la asociación.

Los atributos definen las características de una clase. La clase ***Attribute*** ofrece las propiedades y métodos los determinan cualquier atributo en el modelo. Las propiedades específicas de *Attribute* son:

Containment Indica una relación de contenido.

Derived Define el atributo como derivado.

ExportControl Controla la visibilidad del atributo.

InitValue El valor inicial del atributo.

ParentClass Especifica la clase a la que pertenece el atributo.

Static Define el atributo como estático.

Type Tipo del atributo.

Los objetos de una clase llevan a cabo sus responsabilidades utilizando las operaciones. La clase ***Operation*** ofrece las propiedades y métodos que determinan cualquier operación en el modelo. Las propiedades específicas de *Operation* son:

ReturnType El tipo de retorno de la operación.

Virtual Si la operación es virtual o no.

Parameters La colección de parámetros de la operación.

ExportControl La visibilidad del atributo.

Preconditions El invariante de entrada asumido por la operación.

Semantics La acción de una operación.

Postconditions Los invariantes satisfechos por la operación a la salida.

Protocol El conjunto de operaciones que un cliente puede realizar sobre un objeto.

Qualification Las características específicas a un lenguaje utilizadas para calificar una operación.

Exceptions El conjunto de excepciones que puede disparar una operación.

Size La cantidad absoluta o relativa de almacenamiento utilizado cuando la operación es llamada.

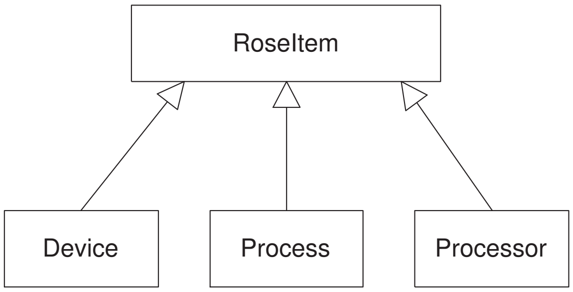

Figura 3.7: Diagrama de herencia de la clase *Deployment*.

Time La cantidad absoluta o relativa de tiempo requerido para completar la operación.

Concurrency La concurrencia de la operación.

ParentClass La clase a la que pertenece la operación.

Y los métodos específicos de *Operation* son:

RemoveAllParameters Eliminar todos los parámetros de la operación.

AddParameter Añadir un parámetro a una operación.

DeleteParameter Eliminar un parámetro de una operación.

Los parámetros califican más aún el comportamiento de una operación. La clase ***Parameter*** ofrece las propiedades y métodos que determinan cualquier parámetro en el modelo. Las propiedades específicas de *Parameter* son:

Const Si el parámetro es constante o no.

Type El tipo de datos del parámetro.

InitValue El valor inicial del parámetro.

3.2.7 Diagrama de herencia de la clase *Deployment*

El diagrama de clases de la figura 3.7 muestra la estructura de herencia de la clase *Deployment* en el modelo REI. Todas las clases heredan de *RoseItem* que pertenece al diagrama de herencia de la clase *Model*.

Se considera un dispositivo como un tipo de hardware que no es capaz de ejecutar un programa. La clase ***Device*** ofrece las propiedades y métodos que determinan cualquier dispositivo en el modelo. Las propiedades específicas de *Device* son:

Characteristics La colección de características del dispositivo.

Y los métodos específicos de *Device* son:

AddDeviceConnection Añadir una conexión (con otro dispositivo) al dispositivo.

AddProcessorConnection Añadir una conexión (con un procesador) al dispositivo.

GetConnectedDevices Obtener la colección de dispositivos conectados al dispositivo.

GetConnectedProcessors Obtener la colección de procesadores conectados al dispositivo.

RemoveDeviceConnection Eliminar una conexión (con otro dispositivo) del dispositivo.

RemoveProcessorConnection Eliminar una conexión (con un procesador) del dispositivo.

Un proceso es la ejecución de un hilo de control en un programa o sistema orientado a objetos. La clase ***Process*** ofrece las propiedades y métodos que determinan cualquier proceso en el modelo. Las propiedades específicas de *Process* son:

MyProcessor El procesador asignado al proceso.

Priority La prioridad del proceso.

Se considera un procesador como un tipo de hardware capaz de ejecutar programas. Los procesadores se asignan para implementar procesos. La clase ***Processor*** ofrece las propiedades y métodos que determinan cualquier procesador en el modelo. Las propiedades específicas de *Processor* son:

Chacteristics Las características del procesador.

Processes La colección de procesos asignados al procesador.

Scheduling El tipo de *scheduling* del procesador (*Preemptive*, *NonPreemptive*, *Cyclic*, *Executive* o *Manual*).

Y los métodos específicos de *Processor* son:

AddDeviceConnection Añadir una conexión (con un dispositivo) al procesador.

AddProcessorConnection Añadir una conexión (con otro procesador) al procesador.

AddProcess Añadir un proceso al procesador.

DeleteProcess Eliminar un proceso del procesador.

GetConnectedDevices Obtener los dispositivos conectados a un procesador.

GetConnectedProcessors Obtener los procesadores conectados a un procesador.

RemoveDeviceConnection Eliminar una conexión (con un dispositivo) del procesador.

RemoveProcessorConnection Eliminar una conexión (con otro procesador) del procesador.

3.2.8 Diagrama de herencia de la clase *Physical*

El diagrama de clases de la figura 3.8 muestra la estructura de herencia de la clase *Physical* en el modelo REI. Las clases *Subsystem* y *Module* heredan de las clases *Package* y *RoseItem* respectivamente, que pertenecen al diagrama de herencia de la clase *Model*.

Un subsistema es una colección de módulos relacionados lógicamente (la relación subsistema/módulo es análogo a la relación categoría/clase). La clase ***Subsystem*** ofrece las propiedades y métodos que determinan cualquier subsistema en el modelo. Las propiedades específicas de *Subsystem* son:

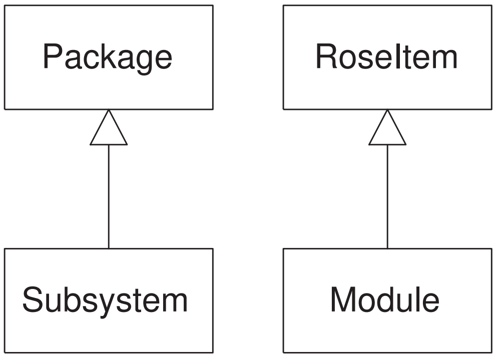

Figura 3.8: Diagrama de herencia de la clase *Physical*.

ModuleDiagrams La colección de los diagramas de módulos asignados al subsistema.

Modules La colección de los módulos pertenecientes al subsistema.

ParentSubSystem El subsistema que contiene al subsistema.

Subsystems La colección de subsistemas hijos del subsistema.

Y los métodos específicos de *Subsystem* son:

AddModuleDiagram Añadir un diagrama de módulo al subsistema.

AddModule Añadir un módulo al subsistema.

AddSubsystem Añadir un subsistema al subsistema.

DeleteModule Eliminar un módulo de un subsistema.

DeleteSubsystem Eliminar un subsistema de un subsistema.

GetAllModules Obtener la colección de módulos pertenecientes al subsistema y a sus descendientes.

GetAllSubsystems Obtener la colección de todos los subsistemas hijos y descendientes.

GetAssignedCategories Obtener la colección de categorías asignadas al subsistema.

GetAssignedClasses Obtener la colección de clases asignadas al subsistema.

GetSubsystemDependencies Obtener las dependencias del subsistema.

GetVisibleSubsystems Obtener los subsistemas visibles.

RelocateModuleDiagram Recolocar un diagrama de módulo a/de un subsistema.

RelocateModule Recolocar un módulo a/de un subsistema.

RelocateSubSystem Recolocar un subsistema a/de un subsistema.

TopLevel Indicar si es el subsistema raíz.

Un módulo es una unidad de código que sirve como un bloque constructivo para la estructura física de un sistema. La clase ***Module*** ofrece las propiedades y métodos que determinan cualquier módulo en el modelo. Las propiedades específicas de *Module* son:

AssignedLanguage El lenguaje de programación asignado al módulo.

Declarations La colección de declaraciones correspondientes al módulo.

OtherPart La segunda parte del módulo cuando forma parte de un subsistema.

ParentSubSystem El subsistema que contiene el módulo.

Part Si el módulo es parte de un subsistema o no.

Path El directorio donde se encuentra el módulo.

Type El tipo del módulo.

Y los métodos específicos de *Module* son:

AddRealizeRel Añadir una relación de realización a un módulo.

AddVisibilityRelationship Crear una nueva relación de visibilidad y añadirla a un módulo.

DeleteRealizeRel Eliminar una relación de realización de un módulo.

DeleteVisibilityRelationship Eliminar una relación de visibilidad de un módulo.

GetAllDependencies Obtener todos los módulos que afectan al módulo.

GetAssignedClasses Obtener la colección de clases asignadas al módulo.

GetDependencies Obtener la colección de módulos incluidos en el módulo.

GetRealizeRelations Obtener la colección de relaciones de realización pertenecientes a un módulo.

GetSubsystemDependencies Obtener la colección de relaciones de visibilidad entre un módulo y un subsistema.

3.2.9 Diagrama de herencia de la clase *Relation*

El diagrama de clases de la figura 3.9 muestra la estructura de herencia de la clase *Relation* en el modelo REI. Estas clases heredan de *RoseItem* que pertenece al diagrama de herencia de la clase *Model*, y son un subconjunto de las clases lógicas del modelo.

Todas las relaciones (*ClassRelation*, *Inherits*, *Has*, *Realizes*) heredan de la clase ***Relation***, cuyas propiedades y métodos permiten especificar y obtener la información sobre los extremos de las relaciones en el modelo. Las propiedades específicas de la clase *Relation* son:

SupplierName El nombre del extremo suministrador de la relación.

Y los métodos específicos de la clase *Relation* son:

HasClient Determinar si la relación tiene una parte cliente.

HasSupplier Determinar si la relación tiene una parte suministradora.

GetClient Obtener el objeto *RoseItem* que actua como cliente de la relación.

GetSupplier Obtener el objeto *RoseItem* que actua como suministrador de la relación.

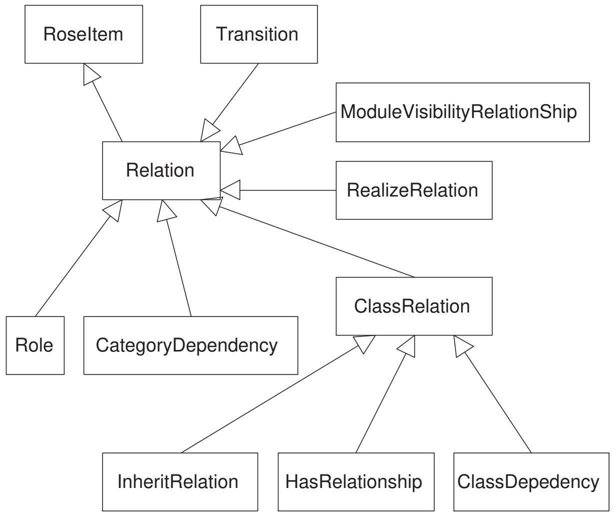

Figura 3.9: Diagrama de herencia de la clase *Relation*.

Los roles denotan el propósito o capacidad en que una clase se asocia con otra. La clase ***Role*** exponen un conjunto de propiedades y métodos para determinar las características de los roles y permiten obtener los roles de un modelo. Las propiedades específicas de la clase *Role* son:

Class La clase correspondiente al rol.

Constraints Las restricciones del rol.

Association La asociación vinculada al rol.

ExportControl La visibilidad de los atributos.

Friend Indica si el rol es "*friend*" o no, permitiendo el acceso a sus atributos y operaciones no públicos.

Containment La relación contenedora del rol.

Cardinality La cardinalidad del rol.

Aggregate Indica si el rol es una clase agregado o no.

Static Indica si el rol es estático o no.

Navigable Indica si el rol es navegable o no.

Keys La colección de claves correspondientes al rol.

Y los métodos específicos de *Role* son:

AddKey Devuelve una clave basada en el nombre y tipo de un atributo.

DeleteKey Elimina una clave de un rol.

GetClassName Obtiene el nombre de la clase asociado con el rol.

La clase ***CategoryDependency*** permite definir y manipular las relaciones de dependencia entre las categorías.

La clase ***ClassRelation*** hereda de la clase *Relation* y es la clase padre de *HasRelationship*, *ClassDependency* e *InheritRelation*. los métodos específicos de *ClassRelation* son:

GetContextClass Obtener la clase cliente de la relación.

GetSupplierClass Obtener la clase suministradora de la relación.

La relación de herencia indica una relación jerárquica entre las clases en las que una clase comparte al estructura y/o comportamiento de otra clase. La clase ***InheritRelation*** expone un conjunto de propiedades y métodos para determinar las características de la relación de herencia. Las propiedades específicas de *InheritRelation* son:

ExportControl La visibilidad de la relación de herencia.

FriendshipRequired Si la propiedad *Friend* es requerida por la relación de herencia.

Virtual Si la relación es virtual o no.

La relación "*Has*" indica una relación contenedora o agregado entre clases. La clase ***HasRelationship*** expone un conjunto de propiedades y métodos para determinar las características de estas relaciones en un modelo. Las propiedades específicas de *HasRelationship* son:

ClientCardinality La cardinalidad del cliente de la relación.

Containment El contenedor de clase para la relación.

ExportControl La visibilidad de la relación.

Static Si la relación es estática o no.

SupplierCardinality La cardinalidad del suministrador de la relación.

La clase ***ClassDependency*** expone un conjunto de propiedades y métodos para determinar las características de las dependencias entre las clases. Las propiedades específicas de *ClassDependency* son:

ClientCardinality La cardinalidad del cliente de la dependencia.

SupplierCardinality La cardinalidad del suministrador de la dependencia.

InvolvesFriendship Si la clase implica la propiedad *Friend* o no.

ExportControl La visibilidad de la dependencia.

Una relación de realización entre una clase lógica y una clase componente muestra que la clase componente realiza las operaciones definidas por la clase lógica. Los métodos específicos de clase ***RealizeRelation*** son:

GetContextClass Obtener la clase cliente de la relación.

GetContextComponent Obtener el componente cliente de la relación.

GetSupplierClass Obtener la clase suministradora de la relación.

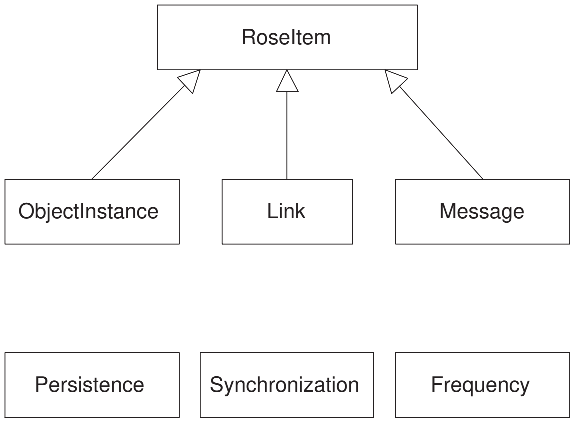

Figura 3.10: Diagrama de herencia de la clase *Scenario*.

GetSupplierComponent Obtener el componente suministrador de la relación.

La clase ***ModuleVisibilityRelationship*** describe el cliente y suministrador de una relación entre módulos. Las propiedades específicas de *ModuleVisibilityRelationship* son:

ContextModule El nombre del módulo cliente de la relación.

ContextSubsystem El nombre del subsistema cliente de la relación.

SupplierModule El nombre del módulo suministrador de la relación.

SupplierSubsystem El nombre del subsistema sumnistrador de la relación.

La clase ***Transition*** es una clase que pertenece al diagrama de herencia de *State*, pero también hereda de la clase *Relation* porque se considera que una transición es una relación entre los estados fuente y destino.

3.2.10 Diagrama de herencia de la clase *Scenario*

El diagrama de clases de la figura 3.10 muestra la estructura de herencia de la clase *Scenario* en el modelo REI. Las clases *ObjectInstance*, *Link* y *Message* heredan de *RoseItem* que pertenece al diagrama de herencia de la clase *Model*.

La clase ***ObjectInstance*** expone un conjunto de propiedades y métodos para determinar las características de los objetos en un modelo. Las propiedades específicas de *ObjectInstance* son:

ClassName El nombre de la clase del objeto.

Links La colección de enlaces correspondientes al objeto.

MultipleInstances Si existen múltiples instancias del objeto o no.

Persistence Si la instancia del objeto es persistente o estática.

Y los métodos específicos de *ObjectInstance* son:

AddLink Añadir un enlace a la instancia del objeto.

DeleteLink Eliminar un enlace de la instancia del objeto.

GetClass Obtener la clase a la que pertenece la instancia del objeto.

IsClass Devuelve $True$ si la instancia del objeto es una clase.

Los objetos interaccionan a través de sus enlaces hacia otros objetos. Un enlace es una instancia de una asociación, de la misma manera que un objeto es una instancia de una clase. Las propiedades y métodos de la clase ***Link*** permiten definir enlaces entre objetos y determinar la naturaleza de las asociaciones. Las propiedades específicas de $Link$ son:

LinkRole1 Define una instancia de objeto como un enlace en $Role1$.

LinkRole2 Define una instancia de objeto como un enlace en $Role2$.

LinkRole1Shared Define una instancia de objeto como un enlace en $Role1$ con visibilidad compartida.

LinkRole2Shared Define una instancia de objeto como un enlace en $Role2$ con visibilidad compartida.

LinkRole1Visibility Define una instancia de objeto como un enlace en $Role1$ y determina su tipo de visibilidad.

LinkRole2Visibility Define una instancia de objeto como un enlace en $Role2$ y determina su tipo de visibilidad.

Y los métodos específicos de $Link$ son:

AddMessageTo Añadir un mensaje al enlace.

AssignAssociation Asignar una asociación al enlace.

DeleteMessage Elimina un mensaje del enlace.

GetMessages Obtener los mensajes llevados por el enlace.

UnAssignAssociation Eliminar una asignación de asociación del enlace.

Los mensajes definen la interacción entre los objetos. La clase ***Message*** ofrece propiedades y métodos que permiten obtener el remitente y destino del mensaje junto a otra información específica del mismo. Las propiedades específicas de $Message$ son:

Frequency La frecuencia del mensaje (si el mensaje es enviado sólo una vez o periódicamente).

Synchronization La semántica de concurrencia del mensaje.

Y los métodos específicos de $Message$ son:

GetSenderObject Obtener el objeto que envió el mensaje.

GetReceiverObject Obtener el objeto que recibió el mensaje.

IsMessageToSelf Si el mensaje es enviado a sí mismo o no.

IsOperation Si el mensaje es una operación o no.

GetOperation La operación asociada con el mensaje.

GetLink Obtener el enlace asociado con el mensaje.

La clase ***Persistence*** define los valores por defecto para la persistencia de objetos. Las propiedades específicas de *Persistence* son:

Persistent El valor por defecto para la persistencia de objetos.

Transient El valor por defecto para la transición de objetos.

La clase ***Synchronization*** define los valores por defecto para la sincronización de objetos. Las propiedades específicas de *Synchronization* son:

Asynchronous Define el valor por defecto para los mensajes asíncronos.

Balking Define el valor por defecto para los mensajes "*Balking*".

Simple Define el valor por defecto para los mensajes simples.

Synchronous Define el valor por defecto para los mensajes síncronos.

Timeout Define el valor por defecto para los mensajes "*Timeout*".

La clase ***Frequency*** define los valores por defecto para la frecuencia de mensajes. Las propiedades por defecto de *Frequency* son:

Aperiodic El valor por defecto para los mensajes no periódicos.

Periodic El valor por defecto para los mensajes periódicos.

3.2.11 Diagrama de herencia de la clase *State*

El diagrama de clases de la figura 3.11 muestra la estructura de herencia de la clase *State* en el modelo REI. Las clases *Action* y *State* heredan de *Element* y *RoseItem* mientras que *Event* y *StateMachine* directamente de *Element*. Tanto *Element* como *RoseItem* pertenecen al diagrama de herencia de la clase *Model*. La clase *Transition* hereda de *Relation*, que pertenece al diagrama de herencia de la clase *Relation*.

Un evento es una ocurrencia que causa una transición de estado. Las propiedades y métodos de la clase ***Event*** definen y controlan eventos que afectan a los estados y establece las transiciones de objetos en un modelo. Las propiedades específicas de *Event* son:

Arguments Las condiciones que afectan al evento.

GuardCondition Define una condición de guarda: cuando sea *True* permitirá la ocurrencia del evento y mientras sea *False* el evento no ocurrirá.

Name El nombre del evento.

Y el método específico de *Event* es:

GetAction Obtener la acción correspondiente al evento.

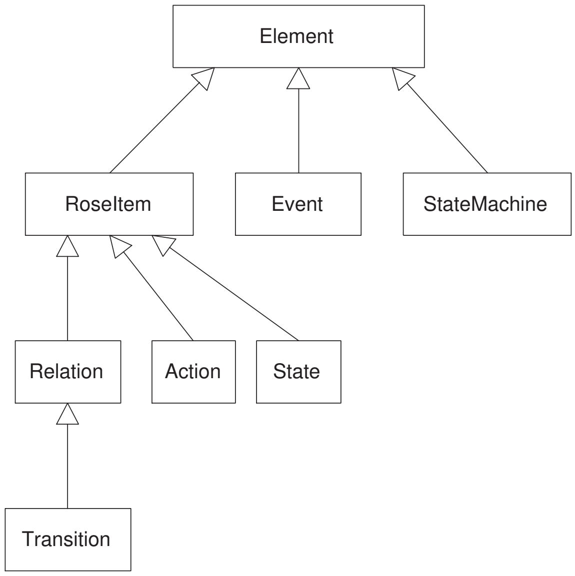

Figura 3.11: Diagrama de herencia de la clase *State*.

La clase ***StateMachine*** contiene toda la información sobre el estado de los objetos de una determinada clase. La máquina de estado de una clase define todos los posibles estados y transiciones para esa clase. Una clase puede tener o no una máquina de estado y una máquina de estado sólo puede pertenecer a una clase. Las propiedades específica de $StateMachine$ son:

Diagram El diagrama de estados que contiene la máquina de estados.

ParentClass La clase a la que pertenece la máquina de estados.

States La colección de estados de la máquina de estados.

Y los métodos específicos de $StateMachine$ son:

AddState Añadir un estado a una máquina de estados.

DeleteState Eliminar un estado de una máquina de estados.

GetAllStates Obtener todos los estados de una máquina de estados.

GetAllTransitions Obtener todas las transiciones de una máquina de estados.

GetTransitions Obtener la colección de transiciones de una máquina de estados.

RelocateState Recolocar un estado en una máquina de estados.

La clase ***Transition*** provee el medio para seguir la traza de un objeto en un cambio de estado, es decir, en el punto donde no está en el estado original pero aún no ha llegado a su estado destino. Los métodos específicos de $Transition$ son:

GetSendAction Obtener el mensaje a enviar cuando la transición ocurre.

GetSourceState Obtener el estado inicial de la transición.

GetTargetState Obtener el estado destino de la transición.

GetTriggerAction Obtener la acción a realizar cuando el evento disparador de la transición ocurra.

GetTriggerEvent Obtener el evento que dispara la transición.

RedirectTo Redireccionar la transición a un nuevo estado destino.

Una acción es una operación que está asociada a una transición, toma una cantidad insignificante de tiempo para realizarse y no se puede interrumpir. Las propiedades de la clase ***Action*** son:

Arguments El contenido de la acción.

Target El objeto destino de la acción.

La clase ***State*** especifica el conjunto de propiedades y métodos que controlan la información sobre el estado de los objetos de un modelo. Las propiedades específicas de la clase *State* son:

History Especifica a qué subestado se vuelve cuando se retorna a un superestado.

ParentState El superestado que contiene al estado.

ParentStateMachine La máquina de estados a la que pertenece el estado.

StateKind El tipo del estado.

Substates Los subestados del estado (por lo tanto, es un superestado).

Transitions El conjunto de transiciones definidas para el estado.

Y los métodos específicos de *State* son:

AddDoAction Añadir una acción ("*Do*") a un estado.

AddEntryAction Añadir una acción de entrada (la primera vez) a un estado

AddExitAction Añadir una acción de salida a un estado.

AddState Añadir un subestado a un estado.

AddTransition Añadir una transición a un estado.

AddUserDefinedEvent Añadir un evento definido por el usuario a un estado.

DeleteAction Eliminar una acción de un estado.

DeleteState Eliminar un subestado de un estado.

DeleteTransition Eliminar una transición de un estado.

DeleteUserDefinedEvent Eliminar un evento definido por el usuario de un estado.

GetAllSubstates Obtener los subestados de un estado.

GetDoActions Obtener las acciones ("*Do*") de un estado.

GetEntryActions Obtener las acciones de entrada de un estado.

GetExitActions Obtener las acciones de salida de un estado.

GetUserDefinedEvents Obtener los eventos definidos por el usuario de un estado.

RelocateState Recolocar un subestado en un estado determinado.

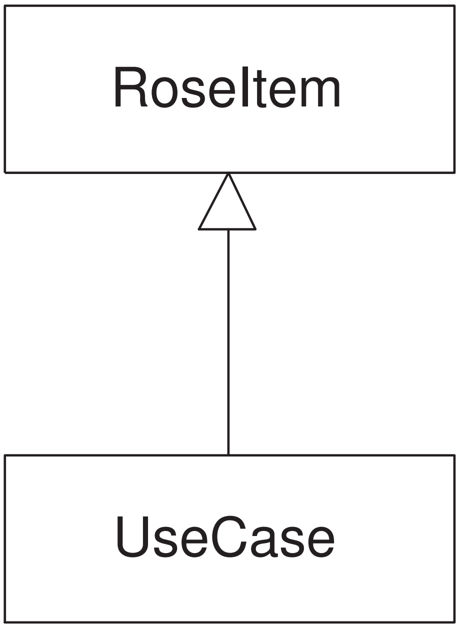

Figura 3.12: Diagrama de herencia de la clase *UseCase*.

3.2.12 Diagrama de herencia de la clase *UseCase*

El diagrama de clases de la figura 3.12 muestra la estructura de herencia de la clase *UseCase* en el modelo REI. La clase *UseCase* hereda de *RoseItem* que pertenece al diagrama de herencia de la clase *Relation*. Se podría pensar en un diagrama de casos de uso en lugar de una clase, pero de esta manera aparecería en el diagrama de herencia de la clase *Diagram*.

La clase ***UseCase*** expone un conjunto de propiedades y métodos que permiten definir y manipular los conjuntos de diagramas de clases y escenarios que forman los casos de uso de un modelo. Las propiedades específicas de *UseCase* son:

Abstract Si el caso de uso es una clase abstracta o no.

ClassDiagrams La colección de diagramas de clases correspondientes al caso de uso.

ParentCategory La categoría que contiene el caso de uso.

Rank El rango del caso de uso

ScenarioDiagrams La colección de diagramas de escenarios correspondientes al caso de uso.

StateMachine La máquina de estado correspondiente al caso de uso.

Y los métodos específicos de *UseCase* son:

AddAssociation Añadir una asociación a un caso de uso.

AddClassDiagram Añadir un diagrama de clases a un caso de uso.

AddInheritRel Añadir una relación de herencia a un caso de uso.

AddScenarioDiagram Añadir un diagrama de escenario a un caso de uso.

DeleteAssociation Eliminar una asociación de un caso de uso.

DeleteClassDiagram Eliminar un diagrama de clases de un caso de uso.

DeleteInheritRel Eliminar una relación de herencia de un caso de uso.

DeleteScenarioDiagram Eliminar un diagrama de escenario de un caso de uso.

GetAssociations Obtener las asociaciones de un caso de uso.

GetInheritRelations Obtener la colección de relaciones de herencia de un caso de uso.

GetRoles Obtener la colección de roles de un caso de uso.

GetSuperUseCases Obtener los super–casos de uso de un caso de uso.

3.3 Diagramas del modelo de empaquetamiento de REI

A continuación se muestran los paquetes en la vista lógica del modelo REI:

Application Classes Application Classes

- Application
- PathMap
- RoseAddInManager
- AddIn
- AddInCollection

Model Classes ModelClasses

Subclasses of Item

Items

Elements

RoseItem

Element

DefaultModelProperties

Subclasses of Relation

Subclasses of ClassRelation

Relations

ControllableUnit

1. Model Collections

- Model
- RoseItem
- ExternalDocument
- ControllableUnit
- Package
- Element
- Property
- DefaultModelProperties

- Relation

View Classes Subclasses of ItemView

Subclasses of Diagram

Diagrams

ItemViews

RoseItemView

- Diagram
- RoseItemView
- NoteView
- View_FillColor
- View_LineColor
- View_Font

Logical Classes Main

Class

ClassDiagram

Category

1. **Related Logical Classes**
   Relationships
   Association
   (a) Undocumented
   - Attribute
   - Operation
   - Parameter
   - Association
   - Role
   - HasRelationship
   - InheritRelation
   - RealizeRelation
   - ClassRelation
   - CategoryDependency
   - ClassDependency
   - CategoryDependencyCollection
   - ...
2. Rich Data Types (Logical)

- Category
- Class
- ClassDiagram
- ClassView

Scenario Classes Scenario Classes

1. Rich Data Types

- ObjectInstance
- Message
- ScenarioDiagram
- Link
- LinkCollection
- InstanceView
- InstanceViewCollection
- Persistence
- Synchronization
- Frequency

Deployment Classes

1. Related Deployment Classes
2. Rich Types (deployment)

- Processor
- Device
- Process
- DeploymentDiagram

Collections Collections

ClassCollection

Subclasses of Collection

1. Undocumented

- ClassViewCollection
- DeploymentDiagramCollection
- DeviceCollection
- ClassDiagramCollection
- ModuleDiagramaCollection
- ClassCollection
- CategoryCollection
- SubsystemCollection
- ModuleCollection
- AttributeCollection
- OperationCollection
- AssociationCollection
- HasRelationshipCollection
- ParameterCollection
- RoleCollection
- InheritRelationCollection
- MessageCollection
- ObjectInstanceCollection
- ProcessorCollection
- PropertyCollection

- ScenarioDiagramCollection
- ItemViewCollection
- ModuleVisibilityRelationshipCollection
- ProcessCollection
- UseCaseCollection
- ExternalDocumentCollection
- NoteViewCollection
- PathMapCollection
- PackageCollection
- ControlableUnitCollection
- ItemCollection
- RealizaRelationCollection

UseCase Classes UseCase

- UseCase

Physical Classes Physical Classes

Module

1. Related Physical Classes

   ModuleVisibilityRelationship

   - ModuleVisibilityRelationship

2. Rich Data Types (Physical)

- Subsystem
- Module
- ModuleDiagram
- ComponentView
- SubsystemView
- ComponentViewCollection
- SubsystemViewCollection

State Classes State Classes

Transition

StateMachine

Action Class

State Classes – Doc

Action

1. Related Classes
2. State View Classes
3. State Collections

- State
- Transition
- Action

- Event
- StateMachine

Base Classes Subclasses of RoseBase

Subclasses of RoseObject

- RoseBase
- RoseObject
- SecondaryRoseObject

Capítulo 4

Formularios

4.1 Definición de formularios

Para definir la plantilla de un formulario se utiliza esta sentencia:

Begin Dialog *DialogName*
  [x],[y],*width*,*height*,*title*$
  [,[.*DlgProc*] [,[*PicName*$] [,*style*]]]
  Dialog Statements
End Dialog

Los formularios también se denominan ventanas de diálogo o cuadros de diálogo para referirse a un pequeño formulario, en general predefinido, para realizar una petición o informar de una determinada acción al usuario. La plantilla de un formulario se construye colocando alguna de las siguientes sentencias entre *Begin Dialog* y *End Dialog* y que definen los controles que formarán el formulario:

- **Picture** Define una imagen.
- **PictureButton** Define un botón con una imagen.
- **OptionButton** Define un botón de opción.
- **OptionGroup** Define un grupo de botones de opción.
- **CancelButton** Define un botón de cancelación.
- **Text** Define un texto no editable.
- **TextBox** Define un control de edición de textos.
- **GroupBox** Define un grupo de controles.
- **DropListBox** Define una lista combinada fija.
- **ListBox** Define una lista.
- **ComboBox** Define una lista combinada.
- **CheckBox** Define un control de verificación.
- **PushButton** Define un botón.

- **OKButton** Define un botón de aceptación.
- **HelpButton** Define un botón de ayuda.

La sentencia *Begin Dialog* necesita los siguientes parámetros:

x, y Coordenadas enteras especificando la posición inicial del formulario.

width, height Coordenadas enteras especificando el tamaño inicial del formulario.

DialogName Nombre de la plantilla del formulario. Una vez que se ha creado la plantilla se puede dimensionar una variable utilizando este nombre.

title$ El nombre que aparecerá en la barra de título del formulario.

.DlgProc Nombre de la función que procesa las acciones en el formulario.

PicName$ Nombre de la DLL que contiene la librería de dibujos.

style Especifica los estilos extras para el formulario y puede tomar algunos de los siguientes valores:

- 0 – El formulario no contiene ni el título ni el recuadro cerrado.
- 1 – El formulario contiene un título pero sin el recuadro cerrado.
- 2 (por defecto) – El formulario contiene tanto el título como el recuadro.

BasicScript genera un error si la plantilla del formulario no contiene algún control y siempre deberá tener al menos uno del tipo *PushButton*, *OkButton* o *CancelButton*. De otra manera, no habrá forma de cerrar el formulario.

4.1.1 La evaluación de expresiones en la plantilla de un formulario

Cualquier expresión o nombre de variable que aparezca en algunas de las sentencias de la plantilla no se evalúa hasta que una variable sea dimensionada para esa plantilla. Estas expresiones no pueden hacer referencia a subrutinas externas o funciones. En el siguiente ejemplo se crea un formulario con el título *Formulario de Ejemplo*:

```
MiTitulo$ = “Hola mundo”

Begin Dialog MiPlantilla 16,32,116,64,MiTitulo$
  OKButton 12,40,40,14
End Dialog

MiTitulo$ = “Formulario de Ejemplo”

Dim UnaVariable As MiPlantilla
rc% = Dialog(UnaVariable)          ’ El título será “formulario de ejemplo”.
```

Todos los controles en un formulario utilizan la misma fuente de letras. Las fuentes utilizadas en los controles de texto pueden cambiarse explícitamente mediante los parámetros adecuados en las sentencias respectivas.

El siguiente ejemplo crea un diálogo de salida:

```
Sub Main()
  Begin Dialog PlantillaDialogoSalida 16,32,116,64,"Salida"
    Text 4,8,108,8,"¿Estás seguro de que quieres salir?"
    CheckBox 32,24,63,8,"Salvar los cambios",.SalvarCambios
    OKButton 12,40,40,14
    CancelButton 60,40,40,14
  End Dialog
  Dim VentanaDialogoSalida As PlantillaDialogoSalida
  rc% = Dialog(VentanaDialogoSalida)
End Sub
```

4.2 Visualizar un formulario

4.2.1 El procedimiento *Dialog*

Se utiliza para visualizar un formulario: no devuelve ningún valor, al contrario que la función del mismo nombre. La sintáxis es la que sigue:

$$Dialog(DialogVariable[,[DefaultButton][,Timeout]])$$

La definición de los tres parámetros es la siguiente:

DialogVariable Nombre de la variable que ha sido previamente declarada como un formulario definido por el usuario.

DefaultButton Un entero que especifica qué botón actúa como botón por defecto en el formulario. Este valor sigue la misma norma que el valor de salida para identificar el botón. El valor por defecto es -1 (Botón OK).

Timeout Un entero que especifica el número de milisegundos para visualizar formulario antes de que automáticamente se quite. El valor por defecto es 0 (sólo se quitará tras una petición explícita del usuario).

En el siguiente ejemplo se visualiza un formulario para mostrar el típico mensaje de error de disco:

```
Sub Main()
  Begin Dialog PlantillaErrorDisco 16,32,152,48,"Error de Disco"
    Text 8,8,100,8,"La disquetera está vacía."
    PushButton 8,24,40,14,"Abortar",.Abortar
    PushButton 56,24,40,14,"Volver a intentar",.Reintentar
    PushButton 104,24,40,14,"Ignorar",.Ignorar
  End Dialog

  Dim ErrorDisco As PlantillaErrorDisco

  Dialog ErrorDisco,3,0
End Sub
```

4.2.2 La función *Dialog*

También se puede utilizar la función *Dialog* que visualiza un formulario definido por el usuario. La función devuelve un entero que representa al botón que se pulsó para salir:

- **-1** El botón *OK* fue pulsado.
- **0** El botón *Cancel* fue pulsado.
- $>$ **0** Se pulsó otro botón: el número representa al botón, según el orden en el que está en la plantilla del cuadro de diálogo.

El siguiente ejemplo muestra de nuevo la ventana a visualizar cuando ocurra un error de disquetera:

```
Sub Main()
  Begin Dialog PlantillaErrorDisco 16,32,152,48,"Error de Disco"
    Text 8,8,100,8,"La disquetera está vacía."
    PushButton 8,24,40,14,"Abortar",.Abortar
    PushButton 56,24,40,14,"Volver a intentar",.Reintentar
    PushButton 104,24,40,14,"Ignorar",.Ignorar
  End Dialog

  Dim ErrorDisco As PlantillaErrorDisco
  r% = Dialog(ErrorDisco,3,0)
  MsgBox "Has seleccionado el botón: " & r%
End Sub
```

4.3 Funciones de mensajes o ventanas de diálogo predefinidas

AnswerBox Muestra un mensaje y espera a que el usuario dé una respuesta, devolviendo un entero indicando qué botón pulsó el usuario.

AskBox, AskBox$ Muestra una ventana para pedir al usuario una entrada de datos, que devuelve como una cadena.

AskPassword, AskPassword$ Muestra una ventana para pedir al usuario una entrada de datos, que devuelve como una cadena. Pero esta vez sólo se visualizan asteriscos cuando el usuario introduzca los datos, permitiendo así la entrada de claves.

InputBox, InputBox$ Muestra una ventana de diálogo con un control de edición de textos para que el usuario pueda introducir algún dato.

MsgBox Muestra un mensaje en una ventana de diálogo con un conjunto de botones predefinidos. Tiene dos versiones: funcional (devolviendo un entero que representa el botón que fue seleccionado) y procedimental.

OpenFilename$ Muestra una ventana de diálogo para abrir un fichero.

SaveFilename$ Muestra una ventana de diálogo para guardar un fichero.

SelectBox Muestra una ventana de diálogo que permite al usuario seleccionar en una lista de posibilidades, devolviendo un entero que representa el índice del elemento que fue seleccionado.

4.4 Métodos y funciones sobre los controles de los formularios

4.4.1 Métodos

ActivateControl Coloca el foco en el control especificado.

DlgCaption Cambia el título de la ventana de diálogo actual.

DlgEnable Habilita/deshabilita el control especificado.

DlgFocus Coloca el foco en el control especificado.

DlgListBoxArray Rellena un control de lista (o variantes) con los elementos de un array.

DlgSetPicture Cambia la imagen contenida en el control (una imagen o un botón con imagen).

DlgText Cambia el texto contenido en el control especificado. El resultado dependerá del tipo de control.

DlgValue Cambia el valor de un control dado. El resultado dependerá del tipo de control.

DlgVisible Muestra u oculta el control especificado.

SelectButton Simula la pulsación de un botón de la ventana.

SelectComboBoxItem Selecciona un ítem de un cuadro combinado de la ventana.

SelectListBoxItem Selecciona un ítem de una lista de la ventana.

SetCheckBox Marca el estado de un control de chequeo de la ventana.

SetEditText Establece el contenido de un editor de texto la ventana.

SetOption Selecciona un botón de opción de la ventana.

4.4.2 Funciones

ButtonEnabled Devuelve *True* si el botón especificado en el formulario activo está habilitado (si es así, puede ser pulsado utilizando la sentencia *SelectButton*).

ButtonExists Devuelve *True* si el botón especificado está presente en el formulario activo.

CheckBoxEnabled Devuelve *True* si el control de verificación especificado en el formulario activo está habilitado (si es así, su valor puede ser establecido utilizando la sentencia *SetCheckBox*).

CheckBoxExists Devuelve *True* si el control de verificación especificado está presente en el formulario activo.

ComboBoxEnabled Devuelve *True* si la lista combinada especificada del formulario activo está habilitada.

ComboBoxExists Devuelve *True* si la lista combinada especificada está presente en el formulario activo.

DlgControlId Devuelve un entero que representa el índice del control especificado tal y como aparece en la plantilla del formulario.

DlgEnable Devuelve *True* si el control especificado está habilitado.

DlgFocus Devuelve una cadena que contiene el nombre del control que tiene el foco.

DlgListBoxArray Rellena una lista, una lista combinada o una lista fija combinada con los elementos de un array, devolviendo un entero que representa el número de elementos que hay actualmente en el control.

DlgText$ Devuelve el contenido textual del control especificado.

DlgValue Devuelve un entero que representa el valor del control especificado. Suele ser el índice del valor seleccionado dentro de un control (listas, grupos de opciones, etc.).

DlgVisible Devuelve *True* si el control especificado está visible.

EditEnable Devuelve *True* si el control de edición de textos especificado está habilitado en el formulario activo.

EditExists Devuelve *True* si el control de edición de textos especificado existe en el formulario activo.

GetCheckBox Devuelve un entero que representa el estado del control de verificación especificado.

GetComboBoxItem$ Devuelve una cadena que contiene el texto de un elemento dentro de una lista combinada.

GetComboBoxItemCount Devuelve un entero que contiene el número de elementos en la lista combinada especificada.

GetEditText$ Devuelve una cadena que es el contenido del control de edición de textos especificado.

GetListBoxItem$ Devuelve una cadena que contiene el elemento especificado en una lista.

GetListBoxItemCount Devuelve un entero que contiene el número de elementos en la lista especificada.

GetOption Devuelve *True* si el botón de opción especificado está activo.

ListBoxEnabled Devuelve *True* si la lista especificada está habilitada en el formulario activo.

ListBoxExists Devuelve *True* si la lista especificada existe en el formulario activo.

MenuItemChecked Devuelve *True* si el elemento de menú especificado existe y su control de verificación está activo.

MenuItemEnabled Devuelve *True* si el elemento de menú especificado existe y está habilitado.

MenuItemExists Devuelve *True* si el elemento de menú especificado existe.

OptionEnabled Devuelve *True* si el botón de opción especificado está habilitado en el formulario activo.

OptionExists Devuelve *True* si el botón de opción especificado existe en el formulario activo.

PopupMenu Muestra un menú emergente que contiene los elementos especificados, devolviendo un entero que representa el índice al elemento seleccionado.

4.5 Procesamiento de acciones sobre un formulario

Cuando se produce una determinada acción sobre un formulario, BasicScript llama a una función especial que tiene asociada y que gestionará el procesamiento de tal acción. La sintáxis para definir esta función es:

$$Function\,DlgProc(ControlName\$, Action, SuppValue)\;As\,Integer$$

Para asociar esta función a un formulario se indica mediante el parámetro *.DlgProc* en la sentencia *Begin Dialog* que lo define. Los parámetros que recibe la función asociada al formulario son:

ControlName$ Cadena que contiene el nombre del control asociado con *Action*.

Action Entero que representa la acción por la que se llamó a la función.

SuppValue Entero que contiene información extra asociada con dicha acción.

Cuando BasicScript visualiza un formulario se producen diversos eventos debidos a la interacción con el usuario: pulsación de botones, introducción de textos por el teclado, selección de elementos de una lista y otros. Cuando se producen estas acciones, BasicScript llama a la función asociada al formulario, pasándole como parámetros: la acción que ha ocurrido, el nombre del control y otra información relevante. La siguiente tabla describe las diferentes acciones que se pueden enviar a una función de formulario:

1 Esta acción es enviada inmediatamente antes de que el formulario se muestre por primera vez: de esta forma se da la oportunidad de preparar la utilización del formulario. Los demás parámetros no son significativos y se ignorará el valor resultado devuelto por la función.

2 Esta acción es enviada cuando:

- Se pulsa un botón. En este caso, $ControlName\$$ contendrá su nombre y $SuppValue$ será: 1 si es un botón Ok, 2 si es un botón $Cancel$ o un valor indefinido si es otro botón. Si la función devuelve 0 como respuesta entonces el formulario será cerrado.
- Se ha modificado el estado de un control de verificación. En este caso, $ControlName\$$ contendrá su nombre y $SuppValue$ el nuevo estado (1 si activo ó 0 si desactivo).
- Se ha seleccionado un botón de opción. En este caso, $ControlName\$$ contendrá su nombre y $SuppValue$ el índice del botón activado dentro del grupo de opciones (que empieza en 0).
- Se ha cambiado la selección actual de una lista o lista combinada. En esta caso, $ControlName\$$ contendrá su nombre y $SuppValue$ el índice del nuevo ítem seleccionado (que empieza en 0).

3 Esta acción es enviada cuando el contenido de un control de edición de texto o una lista combinada ha sido modificado. Esta acción sólo se envía cuando dicho control pierde el foco. En este caso, $ControlName\$$ contendrá su nombre y $SuppValue$ la longitud del nuevo contenido. El valor que devuelve la función es ignorado.

4 Esta acción es enviada cuando un control obtiene el foco de control. $ControlName\$$ contendrá su nombre y $SuppValue$ el índice del control que perdió el foco (que empieza en 0). El valor que devuelve la función es ignorado.

5 Esta acción es enviada continuamente cuando un formulario está inactivo. Si la función devuelve 1 en respuesta, entonces la acción “inactividad” volverá a ser enviada. Pero si la función devuelve 0, entonces la acción no volverá a repetirse. En este caso, $ControlName\$$ no contiene nada y $SuppValue$ el número de veces que se ha enviado hasta el momento.

6 Esta acción es enviada cuando se mueve el formulario. En este caso, los parámetros no son significativos y se ignorará el valor devuelto por la función.

La función de un formulario no puede definir otros formularios, aunque sí podrá invocar cualquier cuadro de diálogo predefinido (tales como $MsgBox$ o $InputBox\$$). Por otro lado, las funciones de los formularios sí podrán utilizar otras sentencias y funciones de BasicScript, permitiendo la manipulación de los controles del formulario de una manera dinámica. Estas son: $DlgVisible$, $DlgText\$$, $DlgText$, $DlgSetPicture$, $DlgListBoxArray$, $DlgFocus$, $DlgEnable$ y $DlgControlId$.

El siguiente ejemplo habilita/deshabilita un grupo de botones de opción cuando un control de verificación es activado/desactivado:

```
Function FuncionFormulario(ControlName$, Action%, SuppValue%)
  If Action% = 2 And ControlName$ = “Imprimir” Then
    DlgEnable “OpcionesImpresion”,SuppValue%
    SampleDlgProc = 1     'No cerrar el formulario.
  End If
End Function

Sub Main()
  Begin Dialog PlantillaFormulario 34,39,106,45,“Ejemplo”,.FuncionFormulario
    OKButton 4,4,40,14
    CancelButton 4,24,40,14
    CheckBox 56,8,38,8,“Imprimir”,.Imprimir
    OptionGroup .OpcionesImpresion
      OptionButton 56,20,51,8,“Apaisado”,.Apaisado
      OptionButton 56,32,40,8,“Vertical”,.Vertical
  End Dialog

  Dim UnFormulario As PlantillaFormulario
  UnFormulario.Printing = 1
  r% = Dialog(UnFormulario)
End Sub
```

Capítulo 5

La gestión de errores

5.1 Introducción

BasciScript ofrece un método para procesar de manera adecuada los errores que ocurren en tiempo de ejecución.

5.1.1 Definición del gestor de errores

Una plantilla genérica para la definición de un gestor de errores dentro de una subrutina podría ser:

```
Sub UnaFuncion()
  On Error Goto catch
  'Código de la función.
  Exit Sub

catch:
  'Código del Procesamiento del Error.
End Sub
```

Brevemente, la línea de ejecución ejecuta el código del algoritmo y si ocurre un error, entonces la línea de ejecución se transfiere al código especial para procesarlo.

5.1.2 La captura de un error

La sentencia *On Error* define la acción a llevar a cabo cuando ocurre un error capturable en tiempo de ejecución. La sintáxis es la siguiente:

On Error {Goto *label* | Resume Next | Goto 0}

La forma *On Error Goto Label* transfiere la línea de ejecución a la etiqueta especificada cuando ocurre un error. La forma *On Error Resume Next* fuerza la continuación de la ejecución por la sentencia que sigue a la causante del error. La forma *On Error Goto* 0 anula cualquier error capturado existente.

5.1.3 El procesamiento del error

Una vez que el gestor del error tiene el control, debería descubrir cuál fue la condición que causó el error y continuar la ejecución con la sentencia *Resume* para reiniciar el gestor de errores, transfiriendo la ejecución a un lugar apropiado dentro del procedimiento actual. La sintáxis es:

Resume {[0] | Next | *label*}

En el siguiente ejemplo se observan varios de sus usos que siguen la siguiente secuencia:

> El usuario teclea dos enteros para multiplicarlos. Si alguno de los números es mayor que un entero, el programa ejecuta una rutina de error y la ejecución del programa continua en una sección específica utilizando *Resume* $< label >$. Entonces se establece un nuevo punto de captura de errores, reiniciando el gestor de errores de la subrutina e indicando que el programa continue su ejecución por la siguiente sentencia incluso si ocurriera algún error.

```
Sub Main()
  Dim a%, b%, x%
OtraVez:
  On Error Goto ErrorRango
  a% = InputBox("Introduzca el primer entero a multiplicar","Introduzca un número")
  b% = InputBox("Introduzca el segundo entero a multiplicar","Introduzca un número")

  On Error Resume Next     'Continua la ejecución del programa en la
  x% = a% * b%             'siguiente línea si ocurre un error.

  if err = 0 then
    MsgBox x%
  else
    Msgbox a% & " * " & b% & " sale del rango de los enteros"
  end if

  Exit Sub

ErrorRango:                'Gestor de errores.
  MsgBox "Has introducido un valor no entero. ¡Inténtalo otra vez!"
  Resume OtraVez
End Sub
```

El error es anulado si el procedimiento termina sin haber ejecutado esta sentencia. En general, existen cuatro formas para reiniciar el gestor de errores de manera que se permite que el procedimiento acabe sin visualizar ningún mensaje de error:

- La ejecución llega a una sentencia *On Error* o *Resume*.
- Se asigna -1 a *Err.Number*.
- Se llama al método *Err.Clear*.
- La ejecución llega a una sentencia *Exit Sub*, *Exit Function*, *End Function* o *End Sub*.

5.1.4 El proceso en cascada para capturar un error

Una característica especial es que se permite el anidamiento de gestores de errores: si ocurre un error dentro del gestor de errores, entonces se llamará al gestor de errores propio.

En cualquier caso si durante la ejecución ocurre un error y no hay definido ningún gestor *On Error* en el procedimiento, entonces BasicScript vuelve al procedimiento que lo llamó y ejecuta allí el gestor de errores.

Este proceso se repite hasta que se encuentra ese gestor, o no haya más procedimientos. Si un error no se captura u ocurre dentro de un gestor, BasicScript muestra un mensaje de error, deteniéndose la ejecución del *script*.

5.2 El objeto predefinido *Err*

Este objeto tiene las siguientes propiedades:

- ***Err.Description*** Establece u obtiene la descripción del error.
- ***Err.HelpContext*** Establece u obtiene el identificador de contexto en la ayuda para conseguir más información sobre el error.
- ***Err.HelpFile*** Establece u obtiene el nombre del fichero de ayuda que contiene la información sobre el error.
- ***Err.LastDLLError*** Obtiene el último error generado por una llamada externa.
- ***Err.Number*** Establece u obtiene el número de error.
- ***Err.Source*** Establece u obtiene la fuente de un error en tiempo de ejecución.

Y ofrece los siguientes métodos:

- ***Err.Clear*** Inicializa las propiedades del objeto *Err*. Esto se realiza automáticamente cuando se ejecuta alguna de las siguientes sentencias: *Resume*, *Exit Function*, *On Error* y *Exit Sub*.
- ***Err.Raise*** Genera un error en tiempo de ejecución, estableciendo las propiedades adecuadas en el objeto *Err* (y que se pasan como parámetros). Éstas son: el número de error, la fuente del error, la descripción del error, el fichero de ayuda con información sobre el error y el contexto dentro de la ayuda.

5.3 Funciones para la gestión de errores

Algunas funciones que van a ayudar para gestionar los errores son:

Erl Devuelve el número de línea (dentro del *script*) del error más reciente.

Error, Error$ Devuelve una cadena que contiene el texto que corresponde al número de error dado o al error más reciente.

IsError Devuelve *True* si la expresión es un valor de error definido por el usuario.

5.4 Compatibilidad con Visual Basic

BasicScript tiene los mismos códigos y mensajes de error que Visual Basic, facilitando la portabilidad entre estos entornos. Ello es debido a que BasicScript obtiene el error mediante la función *Error* o la propiedad *Err.Description*. Hay tres categorías de errores:

1. Errores compatibles con Visual Basic (0 a 799).
2. Errores de BasicScript (800 a 999).
3. Errores definidos por el usuario ($\geq$ 1000).

5.5 Otros ejemplos sobre la gestión de errores

Muestran tres tipos de gestión de un error. El primer caso simplemente ignora el error capturado y continua la ejecución del programa. El segundo caso ejecuta un salto hacia una rutina común de procesamiento de errores, reinicia el gestor y reanuda la ejecución del programa. El tercer caso reinicia el gestor, por lo que la ejecución se parará cuando aparezca el siguiente error.

```
Sub Main()
  Dim x%
  a = 10000
  b = 10000

  On Error Goto Eqit0            'Cuando ocurra un error, salta a esta etiqueta.
  Do
    x% = a * b
  Loop

Etiq0:
  Err = -1                       'Inicializar la señal de error.
  MsgBox "Señar de error inicializada y continuación."

  On Error Goto ErrorRango       'Salto a una nueva rutina de error para
  x% = 1000                      'los siguientes errores.
  x% = a * b
  x% = a / 0

  On Error Goto 0                'Inicializar los saltos de error.
  x% = a * b                     'El programa parará aquí.
  Exit Sub                       'Salida previa a la rutina común de errores.

ErrorRango:                      'Comienzo de la rutina común de errores.
  If Err = 6 then
    MsgBox "Salto por Error en el Rango."
  Else
    MsgBox Error(Err)
  End If

  Resume Next
End Sub
```

Capítulo 6

Otros aspectos del lenguaje de *scripting*

6.1 Descripción de subrutinas y funciones predefinidas

6.1.1 Subrutinas predefinidas

A continuación se muestra una lista de algunas de las subrutinas que ofrece BasicScript. La descripción detallada y muchas otras se encuentran en la ayuda que ofrece Rational Rose.

De las aplicaciones:

AppActivate Activa una aplicación.

AppClose Cierra una aplicación.

AppGetPosition Devuelve la posición de una aplicación.

AppHide Oculta una aplicación.

AppList Rellena un array con los nombres de las aplicaciones abiertas.

AppMaximize Maximiza una aplicación.

AppMinimize Minimiza una aplicación.

AppMove Establece la esquina superior izquierda de una aplicación a una nueva posición.

AppRestore Restaura una aplicación.

AppSetState Maximiza, minimiza o restaura una aplicación, según el valor de un parámetro.

AppShow Hace visible una aplicación.

AppSize Establece la anchura y altura de una aplicación.

Menu Ejecuta una opción de menú de la ventana activa de la aplicación activa.

SendKeys Envía la pulsación de teclas especificadas a la aplicación activa.

De las ventanas de Windows*:*

WinActivate Activa una ventana.

WinClose Cierra una ventana.

WinList Rellena un array con las referencias a todas las ventanas de nivel superior.

WinMaximize Maximiza una ventana.

WinMinimize Minimiza una ventana.

WinMove Mueve una ventana a una nueva posición.

WinRestore Restaura una ventana a un nuevo estado (si estaba minimizada se hace visible y si estaba maximizada vuelve al tamaño anterior).

WinSize Cambia el tamaño de una ventana.

De la administración de ficheros:

ChDir Cambia el directorio actual en la unidad especificada.

ChDrive Cambia la unidad por defecto.

FileCopy Copia de ficheros.

FileDirs Rellena un array de nombres de directorios.

FileList Rellena un array de nombres de ficheros.

Kill Borra todos los ficheros que cumplan una condición específica.

MkDir Crea un nuevo directorio.

Name Renombra un fichero.

RmDir Borra un directorio.

SetAttr Cambia los atributos de acceso de un fichero.

Del sistema:

Date, Date$ Para leer/modificar la fecha del sistema.

DiskDrives Para obtener una lista de unidades del sistema.

DoEvents Devuelve el control al sistema.

DoKeys Simula la pulsación de las teclas especificadas.

Error Simula la ocurrencia de un error de ejecución determinado.

Print Imprime datos en un dispositivo de salida.

PrinterSetOrientation Establece la orientación de la impresora por defecto.

Sleep Para la ejecución del *script* un determinado número de milisegundos.

Stop Finaliza la ejecución del *script*, devolviendo el control al depurador si está activo.

Time Establece la hora del sistema.

Miscelánea:

ActivateControl Pasa el foco directamente a un control.

ArraySort Ordena un array de una dimensión en orden ascendente.

Call Realiza una llamada a una subrutina (los argumentos son opcionales).

Erase Borra los elementos de los arrays especificados.

Otros conjuntos importantes de procedimientos son:

- Procesamiento de cadenas (*Mid*, *Mid$*, *MidB* y *MidB$*).
- Procesamiento de la cola de eventos (*QueEmpty*, *QueFlush*, *QueKeyDn*, *QueKeys*, *QueKeyUp*, *QueMouseClick*, *QueMouseDblClk*, *QueMouseDblDn*, *QueMouseDn*, *QueMouseMove*, *QueMouseMoveBatch*, *QueMouseUp* y *QueSetRelativeWindow*).
- Procesamiento de ficheros de texto y de registro (*Close*, *Get*, *Lock*, *UnLock*, *Open*, *Put*, *Reset* y *Seek*).
- Comunicaciones DDE (*DDEExecute*, *DDESend*, *DDETerminate*, *DDETerminateAll* y *DDETimeOut*).
- Portapapeles (*Clipboard$*).
- Desplazamientos del contenido de una ventana (*HLine*, *HPage*, *HScroll*, *VLine*, *VPage* y *VScroll*).
- Números aleatorios (*Randomize*).
- Registro de Windows y ficheros INI (*DeleteSetting*, *ReadIniSection*, *SaveSetting* y *WriteIni*).
- Asignación especial (Set, LSet y RSet).
- Otras (*Beep*, *Inline* y *MacScript*).

6.1.2 Funciones predefinidas

6.1.3 Funciones

Algunas funciones sobre las aplicaciones:

AppFilename$ Dada una aplicación (dada en los parámetros), devuelve el nombre del fichero que la contiene.

AppFind, AppFind$ Dada una aplicación (dada en los parámetros), devuelve una cadena que contiene el nombre completo de la misma. Se utiliza normalmente para determinar si una aplicación se está ejecutando. Por ejemplo, la siguiente expresión devuelve *True* si "Microsoft Word" se está ejecutando:

$$AppFind\$(\text{``}Microsoft\,Word'')$$

AppGetActive$ Devuelve una cadena que contiene el nombre de la aplicación activa.

AppGetState Devuelve un entero que representa el estado de la ventana especificada (en los parámetros). Estos son *ebMinimized*, *ebMaximized* y *ebRestored*.

AppType Devuelve un entero que representa el tipo de fichero ejecutable de la aplicación especificada (en los parámetros). Estos son *ebDos* y *ebWindows*.

Command, Command$ Devuelve los argumentos en la línea de comandos utilizados para ejecutar la aplicación

DlgCaption Devuelve una cadena que contiene el título del cuadro de diálogo de usuario que está activo.

DoEvents Devuelve el control al sistema operativo, permitiendo a otras aplicaciones seguir su proceso. Si una sentencia *SendKeys* estuviera activa, se espera hasta que todas las teclas en la cola hayan sido procesadas.

Algunas funciones del sistema:

CurDir, CurDir$ Devuelve el directorio actual en el dispositivo especificado.

Date, Date$ Devuelve la fecha actual de sistema.

DateAdd Añade a una fecha un determinado intervalo de tiempo y devuelve el resultado.

DateDiff Devuelve el número de intervalos temporales entre dos fechas.

DatePart Devuelve un entero que representa una parte en una fecha.

DateSerial Devuelve una fecha cuyas partes se pasaron como parámetros.

DateValue Devuelve un *variant* que representa la fecha (que se pasó como parámetro).

Day Devuelve el día del mes de la fecha (que se pasó como parámetro).

DDEInitiate Inicializa un enlace DDE a otra aplicación y devuelve el identificador del canal abierto.

DDERequest, DDERequest$ Obtiene el valor de un ítem de datos a través de un canal DDE abierto.

Dir, Dir$ Devuelve el primer (o siguiente) fichero en un directorio.

DiskFree Devuelve un entero largo que contiene el espacio libre (en bytes) disponible en el dispositivo especificado.

Environ, Environ$ Devuelve el valor de la variable de entorno especificada.

FileDateTime Devuelve un tipo variante fecha (*Date variant*) que representa la fecha y hora de la última modificación de un fichero.

FileExists Devuelve *True* si el nombre de fichero especificado existe.

FileLen Devuelve un entero largo que representa la longitud del fichero especificado en bytes.

FileParse$ Devuelve una cadena que contiene una parte del nombre del fichero especificado, tales como el directorio, la unidad o la extensión del fichero.

FileType Devuelve el tipo del fichero especificado.

GetAttr Devuelve un entero que contiene los atributos del fichero especificado.

Hour Devuelve la hora del día de la fecha/hora especificada.

Mci Ejecuta un comando *Mci* (multimedia).

Minute Devuelve los minutos de la fecha/hora especificada.

Month Devuelve el mes de la fecha/hora especificada.

Now Devuelve un *DateVariant* que representa la fecha y hora actual.

PrinterGetOrientation Devuelve un entero que representa la orientación actual del papel en la impresora por defecto. Los valores pueden ser *ebPortrait* y *ebLandscape*. Esta función carga el *driver* de la impresora y por lo tanto puede ser lenta.

PrintFile Imprime el fichero especificado utilizando la aplicación asociada al mismo.

Random Devuelve un entero largo dentro del rango especificado de manera aleatoria.

Rnd Devuelve un número real simple entre 0 y 1.

Second Devuelve los segundos de la fecha/hora especificada.

Shell Ejecuta una aplicación, devolviendo el identificador de tarea si no hubo errores.

Spc Imprime el número especificado de espacios.

Tab Imprime el número de espacios necesarios para alcanzar la posición de una determinada columna.

Time, Time$ Devuelve la hora de sistema.

Timer Devuelve el número de segundos que han pasado desde medianoche.

TimeSerial Devuelve un *Datevariant* que representa la hora dada con una fecha cero.

TimeValue Devuelve un *Datevariant* que representa la hora contenida en la cadena (pasada como parámetro).

Weekday Devuelve un entero que representa el día de la semana de la fecha especificada.

WinFind Devuelve una variable objeto (HWND) que referencia la ventana que tiene el nombre especificado.

Year Devuelve el año de la fecha especificada.

Algunas funciones matemáticas:

Abs Devuelve el valor absoluto de una expresión.

Atn Devuelve la cotangente de un número.

Cos Devuelve el coseno de un ángulo.

Exp Devuelve el valor de e^n, donde n es un número que se pasa como parámetro.

Fix Devuelve la parte entera del número especificado.

Hex, Hex$ Devuelve una cadena que contiene el equivalente hexadecimal del número especificado.

Int Devuelve la parte entera del número especificado.

Log Devuelve el logaritmo natural de un número.

Oct, Oct$ Devuelve una cadena que contiene el equivalente octal del número especificado.

Sgn Devuelve un entero que representa el signo de un número (mayor, menor o igual a cero).

Sin Devuelve el seno de un ángulo.

Sqr Devuelve la raíz cuadrada de un número.

Tan Devuelve la tangente de un ángulo.

Algunas funciones sobre los arrays:

ArrayDims Devuelve un entero que representa el número de dimensiones del array especificado (en los parámetros).

LBound Devuelve un entero que representa el límite inferior de la dimensión dada de una variable array especificada.

UBound Devuelve un entero que representa el límite superior de la dimensión dada de una variable array especificada.

Algunas funciones sobre las cadenas:

Asc, AscB, AscW Devuelve un entero que representa el código numérico del primer carácter de la cadena (que se pasa como parámetro).

Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ Devuelve el carácter cuyo código se pasa como parámetro.

Format, Format$ Devuelve una cadena formateada según la especificación dada por el usuario.

InStr, InStrB Devuelve la posición de la subcadena dentro de una cadena dadas.

Item$ Dada una cadena que contiene una lista de elementos, devuelve otra cadena con todos aquellos elementos dentro de un rango.

ItemCount Dada una cadena que contiene una lista de elementos, devuelve su número de elementos.

LCase, LCase$ Devuelve la cadena de entrada en letras minúsculas.

Left, Left$, LeftB, LeftB$ Devuelve la subcadena formada por los n primeros caracteres o bytes de la cadena especificada.

Len, LenB Devuelve el número de caracteres o bytes en la expresión de tipo cadena o el número de bytes requeridos para almacenar la variable especificada.

Line$ Dado un texto de entrada, devuelve una o varias líneas que se encuentran entre dos puntos (que también se pasan como parámetros). Se tienen en cuenta aquellos caracteres que especifican habitualmente cambios de línea.

LineCount Devuelve el número de líneas del texto de entrada. Se tienen en cuenta aquellos caracteres que especifican habitualmente cambios de línea.

Mid, Mid$, MidB, MidB$ Devuelve una subcadena de la cadena especificada, dadas una posición inicial y una longitud.

Right, Right$, RightB, RightB$ Devuelve la subcadena formada por los n últimos caracteres o bytes de la cadena especificada.

StrComp Comparación de cadenas.

StrConv Convierte la cadena especificada según un parámetro de conversión: mayúsculas, minúsculas, etc.

String, String$ Devuelve una cadena de una longitud dada que se rellena con el carácter especificado.

Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$ Devuelve una copia de la cadena pasada donde se han eliminado los espacios al inicio o al final de la misma.

UCase, UCase$ Devuelve la cadena de entrada en letras mayúsculas.

Word$ Devuelve una cadena que contiene la secuencia de palabras entre dos puntos dados del texto especificado.

WordCount Devuelve un entero que representa el número de palabras en el texto especificado.

Algunas funciones sobre los tipos:

CBool Convierte una expresión al tipo *Boolean*.

CCur Convierte una expresión al tipo *Currency*.

CDate, CVDate Convierte una expresión al tipo *Date*.

CDbl Convierte una expresión al tipo *Double*.

CInt Convierte una expresión al tipo *Integer*.

CLng Convierte una expresión al tipo *Long*.

CSng Convierte una expresión al tipo *Single*.

CStr Convierte una expresión al tipo *String*.

CVar Convierte una expresión al tipo *Variant*.

CVErr Convierte una expresión en un número de error definido por el usuario.

IsDate Devuelve *True* si la expresión dada puede convertirse a una fecha.

IsEmpty Devuelve *True* si la variable dada es de tipo *Variant* y no ha sido inicializada. Es equivalente a *(VarType(expression) = ebEmpty)*.

IsNull Devuelve *True* si la variable dada es de tipo *Variant* y no contiene datos válidos. Es equivalente a *(VarType(expression) = ebNull)*.

IsNumeric Devuelve *True* si la expresión puede convertirse a un número.

IsObject Devuelve *True* si la variable dada es de tipo *Variant* y contiene un objeto *Object*.

Str, Str$ Convierte un número en una cadena.

TypeName Devuelve el nombre del tipo de la variable especificada.

TypeOf objectvariable Is objecttype Devuelve *True* si *objectvariable* es del tipo *objecttype* especificado. Esta función es muy útil para determinar el tipo de los objetos de automatización OLE.

Val Convierte una determinada expresión cadena a un número.

VarType Devuelve un entero que representa el tipo de dato de la variable especificada.

Algunas funciones que no entran en categorías anteriores:

- Ficheros de textos y binarios (*EOF*, *FileAttr*, *FreeFile*, *Loc*, *Lof* y *Seek*).
- Ficheros INI y el registro de Windows (*GetAllSettings*, *GetSetting*, *ReadIni$*).
- Acceso a base de datos con SQL (*SQlBind*, *SQLClose*, *SQLError*, *SQLExecQuery*, *SQLGetSchema*, *SQLOpen*, *SQLRequest*, *SQLRetrieve* y *SQLRetrieveToFile*).
- Funciones estadísticas (*DDB*, *Fv*, *Ipmt*, *IRR*, *MIRR*, *NPer*, *Npv*, *Pmt*, *PPmt*, *Pv*, *Rate*, *Sln* y *SYD*).
- Portapapeles (Clipboard$).
- Otras (*IMEStatus* y *MacID*).

6.2 Configuración de opciones

La sentencia *Option* configura de manera global al *script* algunos aspectos del procesamiento. Esta sentencia debe aparecer fuera de cualquier bloque de subrutina o función.

El límite inferior de los arrays

Por defecto, el límite inferior es 0, pero se puede poner a 1. La sintáxis que sigue es:

Option Base {0 | 1}

Por ejemplo:

```
Option Base 1

Sub Main()
  Dim a(10)          'Contiene 10 elementos (y no 11).
End Sub
```

La comparación de cadenas

Se puede controlar el modo en que se comparan las cadenas. Por defecto el modo es *Binary* (la comparación es sensitiva a las mayúsculas). Se puede cambiar al modo *Text* (la comparación no es sensitiva). La sintáxis es:

Option Compare [Binary | Text]

Por ejemplo:

```
Option Compare Binary        'Comparación sensitiva a mayúsculas.
```

```
Sub ComparaSensitiva
  a$ = “Esta Cadena Contiene MAYÚSCULAS.”
  b$ = “esta cadena contiene mayúsculas.”

  If a$ = b$ Then
    MsgBox “Las dos cadenas fueron comparadas de manera no sensitiva.”
  Else
    MsgBox “Las dos cadenas fueron comparadas de manera sensitiva.”
  End If
End Sub

Option Compare Text          'Comparación no sensitiva a mayúsculas.

Sub ComparaNoSensitiva
  a$ = “Esta Cadena Contiene MAYÚSCULAS.”
  b$ = “esta cadena contiene mayúsculas.”

  If a$ = b$ Then
    MsgBox “Las dos cadenas fueron comparadas de manera no sensitiva.”
  Else
    MsgBox “Las dos cadenas fueron comparadas de manera sensitiva.”
  End If
End Sub

Sub Main()
  ComparaSensitiva           'Llama a la subrutina definida previamente.
  ComparaNoSensitiva         'Llama a la subrutina definida previamente.
End Sub
```

Las cadenas de escape al estilo de sc C

Habilita o deshabilita la utilización de las secuencias de escape al estilo de C en las cadenas. Cuando está habilitado, el compilador trata el caracter \ como el inicio de una secuencia de escape (los típicos \r, \n, \r, \t, etc.). Por defecto, está opción está deshabilitada. La sintáxis que sigue es:

Option CStrings {On | Off}

Por ejemplo:

```
Option CStrings On

Sub Main()
  MsgBox “Ellos dijeron, \“¡Mira aquello!\””
  MsgBox “Primera línea.\r\nSegunda línea.”
  MsgBox “Carácter A: \x41 \r\n Carácter B: \x42”
End Sub
```

Tipo por defecto de las variables

Establece el tipo por defecto para las variables y funciones cuando no se especifica. Por defecto, este tipo es $Variant$. Actualmente, sólo se puede especificar $Integer$ como tipo alternativo (para mantener compatibilidad con anteriores versiones de BASICSCRIPT. La sintáxis que sigue es:

Option Default *type*

Por ejemplo:

```
Option Default Integer

Function SumaEnteros(a As Integer,b As Integer)
  SumaEnteros = a + b
End Function

Sub Main
  Dim a,b,resultado

  a = InputBox("Teclee un entero:")
  b = InputBox("Teclee otro entero:")

  resultado = SumaEnteros(a,b)
End Sub
```

Sobre la declaración implícita

Se utiliza para prevenir la declaración implícita de las variables y los procedimientos externos. La sintáxis que sigue es:

Option Explicit

6.3 Propiedades y métodos de los objetos predefinidos

6.3.1 El objeto *Basic*

Propiedades

- ***Basic.Architecture$*** Una cadena que contiene la plataforma sobre la que se ejecuta BasicScript. Algunas de ellas son "*Intel*", "*PowerPC*", "*AlphaAXP*", etc. Esta cadena está vacía si la arquitectura no puede ser determinada.

- ***Basic.CodePage*** Un entero que representa la página de códigos de caracteres local.

- ***Basic.Eoln$*** Una cadena que contiene la secuencia de fin de línea apropiada a la plataforma actual.

- ***Basic.FreeMemory*** Un entero largo que representa el número de bytes de memoria libre en el espacio de datos de BasicScript (el tamaño del bloque libre más grande). Previamente, el espacio de datos es compactado para consolidar el espacio libre en un bloque único. Este espacio contiene las cadenas y arrays dinámicos.

- ***Basic.HomeDir$*** Una cadena que contiene el directorio donde se ubica BasicScript.

- ***Basic.Locale$*** Una cadena que contiene información del entorno local donde se está ejecutando BasicScript: formatos de fecha y hora, configuración dependiente del país y análogos.

- ***Basic.OperatingSystem$*** Una cadena que contiene el nombre del sistema operativo. Estos valores son: "*Windows*", "*WindowsforWorkgroups*", "*Win32s*", "*Windows95*", "*WindowsNT*", "*OS/2*", "*Macintosh*" y "*Netware*".

- ***Basic.OperatingSystemVendor$*** Una cadena que contiene el fabricante del sistema operativo sobre el que se ejecuta BasicScript. Estos valores son: "*Microsoft*", "*IBM*", "*Apple*", etc.

- ***Basic.OperatingSystemVersion$*** Una cadena que contiene la versión del sistema operativo sobre el que se ejecuta BasicScript.

- ***Basic.OS*** Un entero indicando la plataforma actual. Estas constantes son: *ebWin16*, *ebWin32*, *ebMacintosh*, *ebSolaris*, etc. Es muy apropiada para la creación de *scripts* multiplataforma.

- ***Basic.PathSeparator$*** Una cadena que contiene el separador en las cadenas de directorio de la plataforma actual.

- ***Basic.Processor$*** Una cadena que contiene el nombre del procesador en el ordenador donde se ejecuta BasicScript. Esta cadena estará vacía si no es capaz de establecerlo.

- ***Basic.ProcessorCount*** Un entero que representa el número de procesadores presentes.

- ***Basic.Version$*** Una cadena que contiene la versión de BasicScript en el formato *major.minor.BuildNumber*.

Métodos

El método ***Basic.Capability*** devuelve *True* si plataforma actual admite la capacidad que se especifica (pasada como parámetro).

6.3.2 El objeto *HWND*

Propiedades

La propiedad por defecto de un objeto *HWND* devuelve un *Variant* que contiene un gestor (*HANDLE*) de la ventana física de una variable objeto *HWND*. Su sintáxis es ***window.****Value*. Esta propiedad es de sólo lectura, y se utiliza para obtener el valor de un objeto *HWND* en el entorno operativo actual. Su tamaño dependerá del sistema en el que se ejecuta el *script* y por lo tanto siempre será guardado en una variable *Variant*.

En el siguiente ejemplo se visualiza una ventana que contiene el nombre de clase de la ventana del *"Program Manager's"* y lo hace utilizando la propiedad *.Value*, pasándola directamente a una rutina externa:

```
Declare Sub GetClassName Lib "user" (ByVal Win%,ByVal ClsName$,ByVal ClsNameLen%)
Sub Main()
  Dim ProgramManager As HWND
  Set ProgramManager = WinFind("Program Manager")
  ClassName$ = Space(40)
  GetClassName ProgramManager.Value,ClassName$,Len(ClassName$)
  MsgBox "El nombre de clase del programa es: " & ClassName$
End Sub
```

6.3.3 El objeto *Msg*

Propiedades

La propiedad ***Msg.Thermometer*** cambia el porcentaje marcado en el termómetro de una ventana de mensaje (abierta previamente con el método *Msg.Open*). Si la ventana no está abierta o el valor está fuera del intervalo 0–100 se genera un error.

Métodos

Los siguientes métodos son específicos para Windows.

- ***Msg.Open*** Muestra una ventana de mensaje con un botón de cancelación y un termómetro que son opcionales. Tiene varios parámetros: el texto del mensaje, el tiempo que se mostrará el mensaje, si aparece el botón de cancelación, si aparece el termómetro y la posición de la ventana. La ventana es no modal, por lo que permanecerá abierta hasta que el usuario seleccione la cancelación, expire el tiempo indicado o se ejecute el método *Msg.Close*. Sólo podrá haber uno de estos mensajes al mismo tiempo y se quitará automáticamente cuando termine el *script*. Hay que llamar periódicamente al procedimiento *DoEvents* para pasar el control al sistema: de esta manera se cierra la ventana cuando se pulse el botón de cancelación.
- ***Msg.Close*** Cierra la ventana de mensaje no modal.

6.3.4 El objeto *Screen*

Propiedades

- ***Screen.DlgBaseUnitsX*** Un entero que se utiliza para realizar la conversión píxeles/unidades de diálogo en el eje horizontal. Este valor dependerá de la fuente utilizada en las ventanas de diálogo. Para pasar de píxeles a unidades:

  $$((XPixels * 4) + (Screen.DlgBaseUnitsX - 1))/Screen.DlgBaseUnitsX$$

  Para pasar de unidades a píxeles:

  $$(XDlgUnits * Screen.DlgBaseUnitsX)/4)$$

- ***Screen.DlgBaseUnitsY*** Un entero que se utiliza para realizar la conversión píxeles/unidades de diálogo en el eje vertical. Este valor dependerá de la fuente utilizada en las ventanas de diálogo. Para pasar de píxeles a unidades:

  $$(YPixels * 8) + (Screen.DlgBaseUnitsY - 1)/Screen.DlgBaseUnitsY$$

  Para pasar de unidades a píxeles:

  $$(YDlgUnits * Screen.DlgBaseUnitsY)/8$$

- ***Screen.Height*** Un entero que representa la altura en píxeles de la pantalla. Es una propiedad de sólo lectura.
- ***Screen.TwipsPerPixelX*** Un entero que representa el número de *twips* por píxel en el eje horizontal del dispositivo de pantalla instalado. La propiedad es de sólo lectura.

- ***Screen.TwipsPerPixelY*** Un entero que representa el número de *twips* por píxel en el eje vertical del dispositivo de pantalla instalado. La propiedad es de sólo lectura.
- ***Screen.Width*** Un entero que representa la anchura en píxeles de la pantalla. Es una propiedad de sólo lectura.

6.3.5 El objeto *System*

Propiedades

Todas ellas son específicas del entorno WINDOWS.

- ***System.FreeMemory*** Un entero largo que indica el número de bytes de memoria libre.
- ***System.FreeResources*** Un entero que representa el porcentaje de recursos libres del sistema. Es un valor entre 0 y 100.
- ***System.TotalMemory*** Un entero largo que representa el número de bytes de la memoria libre disponible.
- ***System.WindowsDirectory$*** Una cadena que contiene el directorio donde se encuentra el sistema operativo.
- ***System.WindowsVersion$*** Una cadena que contiene la versión del sistema operativo.

6.3.6 El objeto *Clipboard*

Métodos

- ***Clipboard.Clear*** Borra el contenido del portapapeles.
- ***Clipboard.GetFormat*** Devuelve *True* si los datos disponibles en el portapapeles siguen un determinado formato (que se pasa como parámetro).
- ***Clipboard.GetText*** Devuelve el texto que contiene el portapapeles.
- ***Clipboard.SetText*** Copia el texto especificado (pasado como parámetro) en el portapapeles.

6.3.7 El objeto *Desktop*

Métodos

Los siguientes métodos son específicos para Windows.

- ***Desktop.ArrangeIcons*** Reorganiza las aplicaciones minimizadas en la pantalla.
- ***Desktop.Cascade*** Reorganiza en cascada las ventanas no minimizadas.
- ***Desktop.SetColors*** Cambia los colores del sistema a un conjunto de colores predefinido (que se pasa como parámetro).
- ***Desktop.SetWallpaper*** Cambia el tapiz de la pantalla al mapa de bits especificado (que se pasa como parámetro).

- ***Desktop.Snapshot*** Toma una sección de la pantalla y la salva en el clipboard como un mapa de bits.

- ***Desktop.Tile*** Reorganiza todas las ventanas no minimizadas de manera que no se superpongan.

6.3.8 El objeto *Net*

Métodos

Los siguientes métodos son específicos para Windows.

- ***Net.CancelCon*** Cancela una conexión de red.

- ***Net.Dialog*** Visualiza una ventana de diálogo que permite configurar la red actualmente instalada.

- ***Net.GetCaps*** Devuelve un entero que informa sobre la red y sus capacidades, según el tipo requerido de información (que se pasa como parámetro).

- ***Net.GetCon$*** Devuelve el nombre del recurso de red asociado al dispositivo local que está redireccionado.

- ***Net.User$*** Devuelve información sobre el usuario conectado a la red.

6.3.9 El objeto *Viewport*

Métodos

Los siguientes métodos son específicos para Windows.

- ***Viewport.Clear*** Limpia la ventana especial abierta.

- ***Viewport.Close*** Cierra la ventana especial abierta.

- ***Viewport.Open*** Abre una ventana especial para mostrar información. Acepta como parámetros el título de la ventana, la posición y el tamaño. En combinación con la sentencia *Print*, estas ventanas son apropiadas para la salida de información de depuración. Esta ventana se cierra automáticamente cuando la aplicación que la abrió termina. Sólo se puede tener una ventana abierta cada vez. El tamaño máximo del búffer que tiene asociado es de 32K., por lo que la información al principio del mismo será borrada para hacer hueco cuando se introduce nuevos datos por el final. Por ejemplo:

  ```
  Sub Main()
    Viewport.Open "Ventana especial de BasicScript",100,100,500,500
    Print "Esto se visualizará en una ventana especial."
    Sleep 2000
    Viewport.Close
  End Sub
  ```

6.4 Directivas de preprocesador

6.4.1 La directiva *#Const*

Define una constante para ser utilizada en la otra directiva del preprocesador #If...Then...#Else. Su sintáxis es:

#Const *constname* = *expression*

Internamente, todas estas constantes son de tipo *Variant*. Por lo tanto, la expresión puede ser de cualquier tipo.

Por ejemplo:

```
#Const SUBPLATFORM = "NT"
#Const MANUFACTURER = "Windows"
#Const TYPE = "Workstation"
#Const PLATFORM = MANUFACTURER & " " & SUBPLATFORM & " " & TYPE

Sub Main()
  #If PLATFORM = "Windows NT Workstation" Then
    MsgBox "Ejecutándose bajo Windows NT Workstation"
  #End If
End Sub
```

6.4.2 La directiva *#If...Then...#Else*

Fuerza al compilador la inclusión o exclusión de secciones del código basándose en unas condiciones dadas. La sintáxis de esta directiva es:

#If *expression* Then
[*statements*]
[#ElseIf *expression* Then
  [*statements*]]
[#Else
  [*statements*]]
#End If

La expresión representa cualquier expresión válida de BasicScript que evalue un tipo booleano. Puede consistir de literales, operadores, constantes definidas con *#Const* o alguna de las constantes predefinidas del preprocesador (por ejemplo, *Win32* es *True* si el entorno de desarrollo está basado en Windows para 32–bits). Cualquier constante no definida evaluará a *Empty*.

Por ejemplo:

```
#Const VERSION = 2
Sub Main
  #If VERSION = 1 Then
    directory$ = "\apps\widget"
  #ElseIf VERSION = 2 Then
    directory$ = "\apps\widget32"
```

```
  #Else
    MsgBox "Versión desconocida."
  #End If
End Sub
```

Una utilización habitual de esta directiva es la inclusión opcional de sentencias de depuración en el código. El siguiente ejemplo muestra cómo se puede depurar el código si se incluye condicionalmente el chequeo de parámetros a una función:

```
#Const DEBUG = 1

Sub CambiarFormato(NuevoFormato As Integer, TextoEstado As String)
  #If DEBUG = 1 Then
    If NuevoFormato <> 1 And NuevoFormato <> 2 Then
      MsgBox "Parámetro " "NuevoFormato" " es inválido."
      Exit Sub
    End If

    If Len(TextoEstado) > 78 Then
      MsgBox "Parámetro " "TextoEstado" " es demasiado largo."
      Exit Sub
    End If
  #End If

  Rem Cambiar el formato aquí...
End Sub
```

El siguiente ejemplo llama a una rutina externa. Estas llamadas son muy específicas de la plataforma, por lo que tendremos que escribir el código adecuado para cada una de ellas, produciendo un código multiplataforma:

```
#If Win16 Then
  Declare Sub GetWindowsDirectory Lib "KERNEL" (ByVal _
    DirName As String,ByVal MaxLen As Integer)
#ElseIf Win32 Then
  Declare Sub GetWindowsDirectory Lib "KERNEL32" Alias _
    "GetWindowsDirectoryA" (ByVal DirName As String,ByVal _
    MaxLen As Long)
#End If

Sub Main()
  Dim NombreDir As String * 256
  GetWindowsDirectory NombreDir,len(NombreDir)
  MsgBox "El directorio de Windows es " & NombreDir
End Sub
```