PROGRAMACIÓN ORIENTADA A OBJETOS EN CLIP

INTRODUCCIÓN AL MODELO OO

Primero que todo, algunas palabras sobre el modelo OO que forma parte de CA-Clipper. Este se basa en un arreglo común y corriente y, cualquier llamada similar a obj:atributo u obj:metodo() , resultará en una situación de búsqueda **secuencial** del primer elemento que coincida con el nombre del atributo o método. Tal búsqueda es ejecutada linealmente y es practicamente análoga a:

ascan(obj, { |x| x[1] == "atributo"})

Esta función reduce grandemente la eficiencia del modelo OO basado en Clipper puro. Esto, por supuesto, no es más que una explicación simplificada, pero el sentido es el mismo que se ha descrito.

Creemos que ahora esta claro, el propósito del por qué optamos por implementar en CLIP un modelo basado en una asociacion del nombre del atributo o método con su posición dentro de un arreglo (como una **indexación directa**, similar a una tabla "hash"). El modelo OO basado en esta asociación con los arreglos de esta forma, es más rápido que el soportado por Clipper.

Al mismo tiempo, no necesitas expresiones similares a ...

obj:=TClass(nombre_clase):new()

en la clase Tclass misma, si no que se recurre a la función "map()" que te crea un objeto vacío asociado a un arreglo. De esta manera, se incrementa la eficiencia del modelo OO.

¿ Cómo podrías crear tu propia clase ? Esto es muy simple de realizar, el ideal es una clase por prg, para poder incluir todas sus funciones de referencia como estáticas :

```
//------------ CREAR CLASE ---------------

function MiClaseNueva()

local obj
obj:=map()                        // Genero un objeto vacío
clone(MyClass2New(),obj)   // Adopto la estructura de MyClass2. De la lib CTI uso "clone".
clone(MyClass3New(),obj)   // Adopto la estructura de MyClass3 (se adiciona)
// Si existen coincidencias de atributos o métodos,
// los elementos de la última clase prevalecen
obj:attribute1:=0
obj:attribute2:=date()
obj:method1:=@func1()       // El "metodo1" se convierte en una referencia a una función
obj:method2:=@func2()
// Estas funciones deben ser definidas en el mismo archivo ".prg" como ESTATICAS.
// Si los métodos han sido adoptados desde otras clases,
// ellos serán reasignados a las clases indicadas

return obj // Retornamos un objeto listo.
```

```
//---------------- FIN ----------------

static function func1()
::attribute1++
return NIL

static function func2(self)
self:attribute1--
return self
```

También, nosotros hemos querido agregar dos simples reglas:

1.- Un atributo es creado cuando algo, incluido NIL, le es asignado.

2.- En cualquier momento en tiempo de ejecución, el método puede asignar o reasignar cualquier función anunciada en este módulo como función estática, o puede adoptar esta función desde otro objeto, como una usual asignación de valores:

```
miObj1:metodo1 := miObj2:metodoX
```

¿ En qué forma pueden ser usados los objetos ? Como en CA-Clipper, o aún mas sencillamente:

```
//----------GENERANDO Y USANDO UN OBJETO ----------

obj := MiClaseNueva()
obj :metodo1()
? obj:atributo1

//-------------------------------------------------------------------------
```

En un objeto, el método "**destroy()**", puede ser utilizado, pero esto no es lo suficientemente destructor, lo cual es usual en lenguajes de tercera generación. Existe una variable local "obj" a la cual se le asigna un objeto. Al abandonar la función, esta variable (como era local) con todos sus datos es destruida. Ahora, consideremos el caso de un objeto que tiene el siguiente atributo:

```
obj:hFile := fopen( "archivo" )
```

Cuando es destruido "obj", es necesario cerrar "hFile", pero el compilador no sabe esto. El compilador (mejor dicho la máquina virtual) sólo conoce que en "hFile" hay un número y sólo destruirá el número, pero el archivo permanecerá abierto. Sólo para tales propósitos se usará un método "mi_destroy()", y éste será llamado (si es que existe) antes de destruir la variable "obj".

```
static function mi_destroy()   //No olvidar, en el mismo prg donde se creó la clase.
fclose(::hFile)
return
```

Y para que el objeto ejecute mi método destructor (presumimos automáticamente al terminar su existencia), debo agregar el método estándar "**destroy()**" y su función referenciada en la creación de la clase. También, se puede reasignar una función posteriormente (ver más adelante).

```
obj:destroy := @mi_destroy()
```

CONTROL DEL CAMBIO DE ATRIBUTOS

Si es necesario controlar cambios de los atributos de un objeto, debe usarse el método "**modify()**" e invocar la función mapmodify(obj, .t.) .

El método modify() es llamado antes de cambiar el valor de cualquier atributo de un objeto. Dos parámetros son pasados a modify(): el código "hash" del atributo a ser cambiado y el nuevo valor a ser asignado. Por ejemplo, modify() debería retornar el valor a asignar al atributo:

```
// En nuestro programa

obj := MyObj()
obj:attr1 := "adios"
? obj:attr1,obj:attr2 // hola mundo


// En el prg de creación de la clase

function MyObj()
local obj := map()
obj:attr1 := ""
obj:attr2 := "mundo"
obj:modify := @mymodify()
mapmodify(obj,.t.)   //Activa (.t.) o desactiva (.f.) el modo de control para el cambio de atributos.
return obj

static function mymodify(self,hash,value)
if hash == hash_ATTR1 .and. value == "adios"
      return "hola" //  Si el atributo vale "adios", le reasigna "hola".
endif
return value
```

RECUPERANDO Y/O REACTIVANDO OBJETOS

CLIP es capaz de almacenar datos de cualquier tipo en los campos MEMO, incluyendo objetos (los convierte en cadenas). Pero no hay forma de almacenar directamente los métodos (las funciones), pues éstos pueden ser cambiados.

Pero existe una manera. Para poder realizar el proceso de recuperación de los objetos con sus correspondientes métodos, se debe proceder de la siguiente forma:

Los datos son decodificados. Si los datos son del tipo objeto y éste tiene el atributo CLASSNAME, entonces la función "*recover&(var:CLASSNAME)(var)"* es llamada. Esta función cumple con reasignar los métodos a este objeto.

Esta característica puede ser usada para enviar objetos como cadenas ("strings") vía correo electrónico o TCP. A continuación un ejemplo de su uso:

```
// Creo el objeto y lo manipulo
x:=asdfNew()

// Visualizo la ejecución de los métodos, veo si funciona.
? "x:m1",x:m1()
? "x:m2",x:m2()

 // Convierto el objeto -> cadena
y:=var2str(x)
// También podría ser: campo->memo_campo:=x

? "y=",y     // Visualizo la cadena resultado de la conversión.

//Reconvierto cadena -> objeto
z:=str2var(y)     // _recover_asdf() es llamada automáticamente
// También podría ser: z:=campo->memo_campo

? "z=",z
? "z:m1",z:m1()     // Verificar si funcionan los métodos
? "z:m2",z:m2()
return


//-------- Creación de la clase  (Esto en un solo prg)--------

function asdfNew()
local o:=map()
o:classname := "ASDF"
o:a1 := "asdf"
o:a2 := "qwer"
_recover_asdf(o)
return o

function _recover_asdf(o)
o:m1 :=@asdf_1()
o:m2 :=@asdf_2()
? "recuperando"
return o

static function asdf_1
? "asdf_1",::a1
return ::a1

static function asdf_2
? "asdf_2",::a2
return ::a1

//------------------- Fin prg -------------------
```

OPERADORES DE SOBRECARGA PARA OBJETOS

CLIP soporta sobrecarga (implementación) de ciertas operaciones con los objetos (lógicas y matemáticas), previa inclusión del método en la creación de la clase. Las operaciones que pueden ser sobrecargadas y sus correspondientes métodos *operador,* son listados en la siguiente tabla :

Operación	Método	Operación	Método
'+'	operator_add	'-'	operator_sub
'*'	operator_mul	'/'	operator_div
'%'	operator_mod	'^'	operator_pow
'|'	operator_or	'&'	operator_and
'$'	operator_in	'='	operator_eq
'=='	operator_eeq	'!='	operator_neq
'<'	operator_lt	'>'	operator_gt
'<='	operator_le	'>='	operator_ge

A continuación, un ejemplo del uso de la sobrecarga de operaciones al usar objetos:

```
//---- En el prg que lo usa ------
oCar1 := newBMW( )
oCar2 := newKAMAZ( )
? oCar1 > oCar2                 // .F. (El “peso” del Kamaz es mayor)
oCar := oCar1 + oCar2
? oCar:model                    //”Scrap”
? oCar:weight                   // 10000
// 10000 kilos del chatarra de fierro.

//----- En el prg que Crea la Clase ----

function newCar()
local obj := map()
obj:model := ""
obj:weight := 0
obj:operator_gt := @car_gt()        //Aquí defino que hacer cuando uso ">"
obj:operator_add := @car_add()   //Aquí defino que hacer cuando uso “+”.
return obj

static function car_gt(car)
return ::weight > car:weight

static function car_add(car)
local obj := newCar()
obj:model := "Scrap"
obj:weight := ::weight+car:weight
return obj

//--------------
```

```
function newBMW( )
local obj := newCar()          // Adopta la clase “Car”
obj:model := "BMW"
obj:weight := 2000
return obj

function newKAMAZ( )
local obj := newCar()          // Adopta la clase “Car”
obj:model := "KAMAZ"
obj:weight := 8000
return obj
```

CONCLUSIÓN

Debido al diseño del modelo OO y su compilación a un programa “C”, existe la posibilidad de escribir las clases estándar Tbrowse y Get en el mismo lenguaje Clipper. Al mismo tiempo, la eficiencia de estas clases no es peor que aquellas escritas en puro “C” para CA-Clipper.