Introducción a la Programación Orientada a Objetos en Fivewin

Parte II

Por: James Bott

Sintaxis de Clases

Antes de escribir sus propias clases, usted necesitará entender la sintaxis de la OOP.

Regla número 1: Usted debe definir cada clase en un PRG propio para ella. Esto es porque los métodos realmente son procesados en funciones estáticas.
Hay tres condiciones de alcance usadas en la definición de una clase: EXPORT (o PUBLIC), PROTECTED (o READONLY) y HIDDEN (o LOCAL).
Todas las variables y métodos están predeterminadas EXPORT de modo que la sintaxis "EXPORT" no es necesaria. Aquí usaremos las condiciones EXPORT, PROTECTED, y HIDDEN.
Debo señalar que en el código de las clases de FiveWin no se usan alcances, por consiguiente, todo es EXPORT. Ésta no es realmente una buena práctica de OOP, porque nada es protegido u oculto, usted puede acceder a los datos y métodos pudiendo causar consecuencias serias.
Como con cualquier código, es importante para la estabilidad hacer clases autosuficientes y tan refinadas como sea posible. Entiéndase que quiero decir que las clases no deben asumir nada, restrinja el alcance de todas las variables tan estrechamente como sea posible, y devuelva cualquier cambio hecho al entorno al estado anterior al finalizar (por ejemplo: áreas de trabajo, punteros de registro, los "SET", etc.).
Usted puede adivinar los significados de las tres condiciones del alcance probablemente, pero repasemos cada una. Por causa de la claridad, aunque no se requiere, yo recomiendo usar el término EXPORT

```
class myclass
 export:
   var itemno
   var descrip
   method new
   method end
endclass
```

La clase anterior se comporta igual con o sin "EXPORT:"Aviso que lleva dos puntos al final (:) y que el término es "EXPORT" no "EXPORTED". Extraño, pero cierto.

Usted puede usar "var" o "data" para definir las variables del caso pero ya que el propio código fuente de FW usa DATA nosotros probablemente debamos apegarnos a esto.
También pueden definirse las variables como "private" (privadas) o "readonly" (solo para lectura). Las variables son "readonly" cuando se accede a ellas desde fuera de la clase pero pueden cambiarse desde dentro de la clase.

```
class myclass
   data itemno
   data onHand readonly
endclass
```

Lo anterior puede reescribirse de esta manera sin cambio en la funcionalidad:

```
class myclass
 export:
   data itemno
 protected:
   data onHand
endclass
```

Yo prefiero el segundo estilo, de nuevo por la claridad.
Note que pueden modificarse las variables protegidas desde dentro de la clase y también dentro de cualquier subclase de la misma.
Aunque usted puede definir un método como PROTECTED, no es necesario ya que los métodos son "READONLY" de todos modos. Los métodos pueden ser visibles fuera de la clase o no, ellos serán visibles si se definieron como "export" o "protected".
Las variables y métodos pueden ser HIDDEN que significa que no son visibles fuera de la clase. Según el principio de ocultamiento de la OOP de información, usted debe esconder cualquier dato y método que sólo sean necesarios para el uso interno de la clase. La ventaja de esto es que usted puede modificar la entrañas de su clase sin ninguna preocupación de que otro código pueda estar llamándola. Usted puede cambiar la sintaxis, etc.,

```
class myclass
 export:
   data itemno
 hidden:
   method calcReorder()
endclass
```

Algo más que podría querer hacer por la claridad del código es definir el tipo de sus variables.

```
class myclass
 export:
   data itemNo as char
 protected:
   data onHand as number
 hidden:
   data oCustomer as object
endclas
```

O, usando la notación húngara que hace el tipo de datos auto explicado

```
class myclass
 export:
   data cItemNo
 protected:
   data nOnHand
 hidden:
   data oCustomer
endclass
```

Por una razón que discutiremos después, esto no siempre es factible.

Los métodos son simplemente funciones estáticas, a fin de que pueda devolver variables desde ellos. Usted también puede devolver "SELF" que simplemente refiere al mismo objeto. Devolver "self" le permite encadenar las llamadas del método en una sola línea así:

```
oCustomer:new():edit()
```

Usted verá cuan práctico es más tarde.

A veces querrá usar una palabra reservada de Clipper para nombre de un método, por ejemplo, "delete".
Si usted intenta:

```
method delete
```

obtendrá un mensaje del error. Hay una manera para esto, Usted define el método usando la cláusula MESSAGE y compone un nombre diferente para el método real.

```
class Customer
   message delete method _delete
endclass
...
```

Entonces escriba el método como sigue:

```
class Customer
    message delete method _delete
endclass

method _delete()
...
```

Ahora usted puede llamarlo usando "delete" :

```
oCustomer:delete ()
```

Diseño de Clases:
El propio código fuente de Fivewin está totalmente consagrado a la manipulación de datos y diseño de interfaz. Nosotros podemos escribir subclases para modificar la conducta predeterminada de FW para mejora nuestras aplicaciones y/o nosotros podemos escribir las nuevas clases para agregar las características adicionales.
Para crear una clase que herede de una clase existente simplemente use la cláusula FROM.

```
class myClass from TWhatever
   method new
   method balance
endclass
```

Lo anterior crea una nueva clase de la clase existente TWhatever. Cada método definido o es un nuevo método o reemplaza el método de la clase madre.

Clase de aplicación

Si usted escribe muchos programas diferentes, una herramienta muy útil para tener sería una clase de la aplicación. Esto sirve como esqueleto de arranque para una nueva aplicación. Se encarga de la configuración básica de su aplicación.
Lo que sigue es un ejemplo simple de una clase de aplicación. Úsela como PRG principal. Todo lo que tiene que hacer es rellenar el BuidlMenu () y BuildBar () los métodos para tener un buen principio en una nueva aplicación.

```
/ / La Clase de la aplicación  //
#include "fivewin.ch"
#define APP_RESOURCES       "myapp.dll"
#define APP_HANDLECOUNT     120
#define APP_HELPFILE    "myapp.hlp"
function main()
   TApp():new():activate()
return nil
class TApp
   export:
      data oWnd,oMenu,oBar,oFont
      method new
      method activate
   hidden:
      method buildMenu
      method buildBar
endclass
method new()
   set epoch to 1980
   set deleted on
   set default to (cFilePath( GetModuleFileName( GetInstance() ) ))
   set resources to APP_RESOURCES
   setHandleCount(APP_HANDLECOUNT)
   setHelpFile(APP_HELPFILE)
   define font ::oFont name "MS Sans Serif" size 0,-12
return self
method activate()
   define window ::oWnd MDI;
      menu ::buildMenu()
      ::buildBar()
      set message of ::oWnd to "My application"
   activate window ::oWnd
   set resources to
   close databases
   ::oFont:end()
Return nil
method buildMenu()
   menu ::oMenu
      menuItem "&File"
         menu
            menuItem "Exit  Alt-F4" action ::oWnd:end()
         endmenu
      menuItem "&Edit"
   endmenu
return ::oMenu
method buildBar()
   define buttonbar ::oBar of ::oWnd size 26,28 3d
   define button of ::oBar
return ::oBar
```

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

Clases Comerciales

Donde personalmente, pienso que podemos sacar el máximo provecho de la OOP es en el desarrollo y uso de clases comerciales. Una de las ventajas de usar OOP es que nosotros podemos fácilmente eliminar la distancia entre los usuarios y desarrolladores usando una sintaxis en común para describir el dominio. En esencia, nuestra meta es primero desarrollar un modelo de la situación comercial, entonces construir este modelo utilizando OOP.
Para describir el dominio, nosotros necesitamos sacarnos nuestro sombrero de programador para intentar describir el negocio en términos de objetos del mundo real, como cliente, el artículo, orden, factura, etc., En esta mezcla nosotros podemos agregar también a los objetos de proceso como: hago-una-cosa, proceso-una-orden, etc., Es interesante notar que éste no es ningún pensamiento nuevo; Aristóteles describió esto en su monografía," Categorías" en 350 AC.
Como libro muy bueno sobre ingeniosas clases comerciales, vea “La Ingeniería Comercial con la Tecnología del Objeto” disponible en mi sitio web. Yo uso mucho este libro. No es técnico, en eso no hay ningún ejemplo del código, pero cubre la filosofía y diseño de objetos de clases comerciales.
Como programadores nosotros hemos hecho ya alguna clasificación cuando reunimos datos en archivos de cliente, artículo, etc., Estos archivos contienen los DATOS de un objeto del mundo real del mismo nombre. Lo que no contienen son los métodos. Afortunadamente, FW nos proporciona las bases para una buena solución al crear los objetos comerciales desde archivos de datos con su clase TDatabase. Esta clase tiene la capacidad curiosa de construir una definición de si misma al vuelo.
Cuando el DBF se abre todos los nombres de los campos se vuelven DATOS de la clase automáticamente. Nosotros podríamos codificar esto a mano en una definición de la clase, pero TDatabase maneja esto automáticamente para nosotros.
TDatabase también agregó los métodos para todas las funciones comunes de manejo de bases de datos como buscar, saltar al siguiente, etc. Uno de los rasgos más importantes que agrega es un buffer de datos. Todos los campos de datos son automáticamente cargados en un array al que accedemos usando los nombres de las variables de la clase:

```
oCustomer:name
```

Esto refiere en realidad a oCustomer:aBuffer[x] donde x es la posición en el array aBuffer donde se guardan los nombres de los datos. Esto significa que nosotros no tenemos que preocuparnos más por usar rutinas de esparcir y juntar.
Hay un pequeño conflicto que nosotros necesitamos discutir. Un objeto “Cliente” (Customer en inglés) se refiere propiamente a un solo cliente, entonces un objeto “Clientes” (en plural) referirá a la tabla (base de datos) de clientes. Uno podría ver esta diferencia como un registro vs. una base de datos. Un objeto oCustomer contendría todos los DATOS como el nombre, dirección, etc, y los métodos cargar() (load() en inglés) y guardar() (save() en inglés) y otros métodos del estilo. Un objeto de tabla oCustomers tendría métodos de navegación, métodos de indizado y del estilo. No tendría la necesidad de métodos cargar() y guardar() porque de aquéllos se ocuparía el objeto oCustomer (singular).
OK, entonces un objeto oCustomer sería diferente que un objeto oCustomers. Pero la clase TDatabase de FW no está diseñada usando un objeto registro. Contiene todos los DATOS de un objeto del oCustomer y también todos los métodos del metodo tabla oCustomers. Toda una mezcla.

Para separar un objeto oCustomer singular lo que nosotros necesitamos hacer es copiar todos los datos del registro del dbf a un nuevo objeto algo así:

```
class customer
 export:
   data custNo
   data name
   data address
   data city
   data state
   method new
   method load
   method save
 hidden:
   oCustomers
endclass
method new(cCustNo,oCustomers)
   ::custno:= cCustNo
   ::oCustomers:= oCustomers
return self
method load()
   ::oCustomers:seek(::custno)
   ::name    := ::oCustomers:name
   ::address := ::oCustomers:address
   ::city    := ::oCustomers:city
   ::state   := ::oCustomers:state
return self
method save()
   ::oCustomers:seek(::custno)
   ::oCustomers:name    := ::name
   ::oCustomers:address := ::address
   ::oCustomers:city    := ::city
   ::oCustomers:state   := ::state
   ::oCustomers:save()
return self
```

Esto separa el cliente singular, oCustomer muy bien de la tabla oCustomers. Hay algunos problemas de fondo: Primero, esto requiere codificación extra y requerirá mantenimiento cada vez que un campo se agregue o cambie en el DBF. Habrá también una pequeña pérdida en el desempeño que puede o no volverse un problema significativo.
Yo prefiero usar simplemente el objeto de tabla para ambos el cliente plural y singular. Esto es más simple y más rápido. Por convención, doy siempre un nombre singular al objeto tabla, como oCustomer para que el referirse a los datos parezca más natural:

```
oCustomer:name
```

Esta no es una buena práctica de la OOP, pero hace las cosas más fáciles y eficaces.
Simplemente recuerda que estas clases son una combinación de objetos de clase comercial y de tabla
TDatabase tiene carencias en algunas áreas, y las he tratado mediante el desarrollo de una subclase que llamo TData. Éste es un producto del shareware que está disponible en mi sitio web. La versión no registrada es un OBJ totalmente operativo que contiene una simple pantalla. El código fuente completo se proporciona al registrarse. Obteniendo la demo de TData le permitirán probar todo el código que sigue.

Un agregado importante proporcionada por TData es la generación automática de nombres de alias. Cuando usted crea un nuevo objeto este se ocupa del alias automáticamente. Esto le permite abrir multiples copias del mismo DBF sin preocuparse por los alias.
Encuentro a TData inestimable al implementar clases comerciales.
Pero antes, entremos en lo necesario para discutir algunos problemas básicos de entre casa: Usted necesitará crear un subclase de TData para cada uno de sus archivos de base de datos.

Primero defina la clase:

```
class TCustomer from TData
export:
   data cTitle init "Customer"
   method new
   method add
   method edit
   method browse
endclass
```

La primera cosa que querrá hacer es poner el código para apertura de los índices en el nuevo método (encapsulandolo en su clase):

```
method new()
   super:new(,"cust")
   ::use()
   if ::used()
      addIndex("cust")
      addIndex("cust2")
      ::setOrder(1)
      ::gotop()
   endif
return self
```

Ahora, siempre que usted necesite usar este DBF todo lo que necesita es:

```
oCustomer:= TCustomer():new()
```

Esto abre el archivo cust.dbf y todos sus índices y mueve el puntero al primer elemento del primer índice.

Si alguna vez necesita agregar un índice, solo tendrá que agregar el código al método new() (nuevo en inglés) y se modificará automáticamente en todas partes. Usted también podría cambiar el nombre de archivo o nombres del índice sin preocuparse como afecte su programa. Ésta es la ventaja de la “información oculta” uno de los conceptos que promueve la OOP.
Ahora necesitamos agregar funciones básicas para estas clases para objetos comerciales. Cada una necesitará métodos para agregar y editar. Si usted está creando una aplicación MDI también puede agregar un método “browse” (vista) genérico.
El método Add() (agregar en Inglés) es bastante simple. Puede usar este método para entrar datos, solo necesita poner en blanco el buffer y poner una bandera para que el método Edit() (modificar) sepa a fin de que añada un nuevo registro.

```
method add()
   ::lAdd:=.t.
   ::blank()
   ::edit()
   ::lAdd:=.f.
   ::load() // En caso de cancelar, vuelve a cargar el buffer
return self
```

Cree un metodo Edit() (editar) que simplemente contenga una ventana de edición normal. En la cláusula ACTION manejamos el append (adición) si es un nuevo registro.

```
method edit(oWnd)
   local oDlg
   default oWnd:= wndMain()
   define dialog oDlg resource "charge" of oWnd title ::cTitle
      redefine get ::date ID 107 of oDlg update
      redefine get ::custno ID 108 of oDlg update
      redefine get ::company ID 109 of oDlg update
      redefine get ::address ID 110 of oDlg update
      redefine get ::phone ID 104 of oDlg update
      redefine button ID 1 of oDlg action ;
        (if(::lAdd,::append(),),::save(),oDlg:end())
   activate dialog oDlg centered
return self
```

Ahora nosotros agregamos un browse (una vista) genérico en una ventana MDI hija. Aquí estamos usando simplemente la cláusula FIELDS que despliega todos los campos. Note que cuando la ventana del browse se cierra llamamos al método ::End(), método que cierra la base de datos (como la declaración CLOSE). Hacemos esto para que podamos llamar un browse con una línea (hay un ejemplo después).

```
method browse()
   local oWnd, oLbx
   define window oWnd mdichild of wndMain() title ::cTitle
      @ 0,0 listbox oLbx fields alias ::cAlias of oWnd update;
        font appFont()
      oLbx:bSkip:={|nRecs| ::Skipper(nRecs) }
      oWnd:setControl(oLbx)
   activate window oWnd maximized valid (oLbx:cAlias:="",::end())
return self
```

Vea cuidadosamente los métodos Add() y Edit(). Verá que los dos son completamente genéricos, pueden usarse para cualquier clase de base de datos. ¡Ah ha! Éstos son los candidatos perfectos para una clase madre. Ambos métodos serían heredados por cualquier clase hija. Creemos una nueva clase hija de TData que usamos como madre de clases de base de datos. Primero veamos el árbol de herencia:

```
class TDatabase
class TData from TDatabase
class TXData from TData
class TCustomer from TXData
```

Aquí está la nueva clase de TXData. Agregamos un metodo Edit() virtual para que el compilador no se queje ya que el metodo Edit() se llama en el método Add(). Este método virtual no hace

nada.

```
class TXData from TData
 export:
   method add
   method edit virtual
   method browse
endclass
method add()
   ::lAdd:=.t.
   ::blank()
   ::edit()
   ::lAdd:=.f.
   ::load() / / en caso de que cancelen, recarga el buffer
return self
method browse()
   local oWnd, oLbx
   define window oWnd mdichild of wndMain() title ::cTitle
      @ 0,0 listbox oLbx fields alias ::cAlias of oWnd update font;
            appFont()
      oLbx:bSkip:={|nRecs| ::Skipper(nRecs) }
      oWnd:setControl(oLbx)
   activate window oWnd maximized valid (oLbx:cAlias:="",::end())
return self
```

Ahora nuestra clase de TCustomer consiste en sólo dos métodos:

```
class TCustomer from TXData
   export:
   method new
   method edit
endclass
method new()
   super:new(,"cust")
   ::use()
   if ::used()
      addIndex("cust")
      addIndex("cust2")
   endif
   ::setOrder(1)
   ::gotop()
return self
method edit(oWnd)
   local oDlg
   default oWnd:= wndMain()
   define dialog oDlg resource "charge" of oWnd title ::cTitle
      redefine get ::date ID 107 of oDlg update
      redefine get ::custno ID 108 of oDlg update
      redefine get ::company ID 109 of oDlg update
      redefine get ::address ID 110 of oDlg update
      redefine get ::phone ID 104 of oDlg update
      redefine button ID 1 of oDlg action ;
               (if(::lAdd,::append(),),::save(),oDlg:end())
   activate dialog oDlg centered
return self
```

Usted puede heredar todas sus bases de datos desde TXData y sólo tiene que definir los métodos New() y Edit(). Claro, el método browse() es genérico por lo que solo será útil como prototipo, probablemente querrá escribir un nuevo método Browse() personalizado por cada base de datos.

Lo bueno de tener otra clase entre TData y sus clases de la base de datos específicas es que usted puede agregar otras cosas que se heredarán a todas sus clases. Digamos que quiere guardar respaldo de la última fecha y hora en que cualquier registro fue actualizado. Usted necesitará tener un campo llamado "Updated" (actualizado) en cada una de sus bases de datos. Entonces simplemente defina un nuevo método Save() (guardar) en TXData:

```
method save()
   ::updated:= dtoc(date())+" "+time()
return super:save()
```

¡Ahora cada vez que se guarda un registro, el campo "Updated" mostrará la fecha y hora de dicha actualización!
Anteriormente le llamé la atención sobre el ::End() en el método Browse(). Esto nos permite asociar este breve código a un botón o menú:

```
DEFINE BUTTON of oBar ACION TCustomer:new():browse()
```

Esto abrirá una ventana MDI hija con un browse de la base de datos de clientes en él. Cuando la ventana está cerrada, la base de datos también está cerrada.

Agregando Métodos Comerciales

Ok, hasta ahora todo lo que nosotros hemos discutido tiene que hacer con la interfaz del usuario. El tema realmente excitante es los métodos comerciales. Es con estos métodos conque nosotros reflejamos la conducta de estos objetos en el mundo real. Por ejemplo, en una clase cliente nosotros podríamos tener un método acceptPayment () (recibir pago)

```
method acceptPayment(nPayment) class TCustomer
  ::balance:= ::balance - nPayment
  ::lastPayment := nPayment
  ::save()
return nil
```

O, usted podría querer también actualizar cualquier importe de factura:

```
method acceptPayment(nPayment) class TCustomer
  ::balance:= ::balance - nPayment
  ::lastPayment:= nPayment
  ::save()
  oInvoice:=TInvoice():new()
  oInvoice:acceptPayment(self, nPayment)
  oInvoice:end()
return nil
```

Ahora aplicar un pago a un objeto oCustomer (cliente) todo lo que necesita es:

```
oCustomer:acceptPayment(nPayment)
```

Quizás usted comienza a ver el poder de la OOP y los objetos de negocio.

Resumen

Yo creo firmemente que usar los conceptos de la OOP en su programación lo llevará a todo un nuevo nivel de productividad, además, sus programas serán mucho más estables, y los cambios y agregados se harán mucho más fáciles. Usted nunca lamentará haber dedicado tiempo a aprender estos conceptos.

 Escrito por James Bott, jbott@compuserve.com http://ourworld.compuserve.com/homepages/jbott,