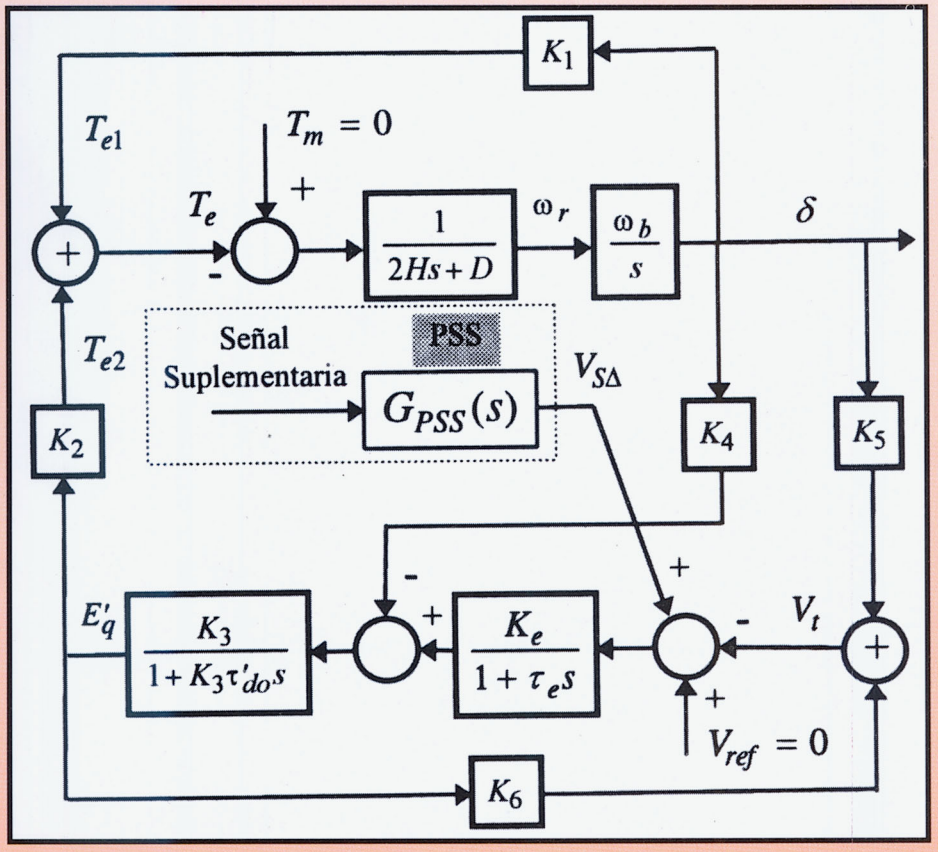
ISSN 1409-2441

Ingeniería

Revista de la Universidad de Costa Rica
Enero/Junio 1997 VOLUMEN 7 N° 1



ISSN 1409-2441

INGENIERIA

Revista Semestral de la Universidad de Costa Rica
Volumen 7, Enero/Junio 1997 Número 1

CORRESPONDENCIA Y SUSCRIPCIONES

Editorial de la Universidad de Costa Rica
Apartado Postal 75
2060 Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

CANJES

Universidad de Costa Rica
Sistema de Bibliotecas, Documentación e Información
Unidad de Selección y Aquisiciones-CANJE
Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

Suscripción anual:
Costa Rica: ¢ 1 000,00
Otros países: US $ 25,00

Número suelto:
Costa Rica: ¢ 750,00
Otros países: $ 15,00

Edición aprobada por la Comisión Editorial de la Universidad de Costa Rica

Ciudad Universitaria Rodrigo Facio
San José, Costa Rica.

Revisión Filológica: *Lorena Rodríguez*

Diseño Gráfico, Diagramación y Control de Calidad:
Sergio Aguilar Mora

Impreso en la Oficina de Publicaciones
de la Universidad de Costa Rica

Revista
620.005
I-46i Ingeniería / Universidad de Costa Rica. ——
Vol. I, no. 1 (ene./jun. 1991)— . —— San José, C. R. : Editorial de la Universidad de Costa Rica, 1991— (Oficina de Publicaciones de la Universidad de Costa Rica)
v. : il

ISSN 1409-2441

Semestral.

1. Ingeniería - Publicaciones periódicas.

CCC/BUCR—250

Ingeniería Vol. 7 Nº 1 - 1997: 53-64; San José, Costa Rica

¿ES VISUAL FOXPRO UN AMBIENTE ORIENTADO A OBJETOS?

Vladimir Lara V. *
Maureen Murillo R. **

RESUMEN

En este artículo, mediante el desarrollo de un ejemplo, se describen y analizan las características de Visual FoxPro relacionadas con la orientación a objetos, para determinar si es un ambiente orientado a objetos según las definiciones descritas en el artículo "¿Es un ambiente orientado a objetos?" Lara y Murillo, 1997.

SUMMARY

With an example, this paper describes and explains the features of Visual FoxPro in relation to object orientation, in order to determine if it is an object-oriented programming environment according to the definitions introduced in the paper "¿Es un ambiente orientado a objetos?" Lara y Murillo, 1997.

1. Introducción

Para evaluar si Visual FoxPro (VFP) es un ambiente orientado a objetos, este artículo se basará en la definición de los elementos que están asociados al concepto de orientación a objetos que se hace en el artículo "¿Es un ambiente orientado a objetos?" Lara y Murillo, 1997. Este artículo clasifica estos elementos, según su función, como características determinantes en la orientación a objetos de un ambiente. La clasificación define tres grupos de aspectos: los que definen las características básicas que debe poseer un lenguaje para ser considerado como orientado a objetos, los que pueden ser considerados como las características asociadas a un lenguaje orientado a objetos y las facilidades que ofrece un ambiente para la manipulación de los objetos.

Para la descripción de las distintas características de VFP se utilizará a lo largo de este artículo el ejemplo presentado en la figura N°1. Esta figura muestra la ventana de diseño de formas (pantallas de interfaz) de VFP con una forma ya diseñada. Esta forma corresponde al diseño de un plan de estudio de una carrera universitaria de computación. El diseñador de formas muestra los diferentes objetos que forman la ventana (títulos, botones, cajas de texto, etc.), tal y como serán observados por el usuario. El objetivo de esta forma es que el usuario pueda definir relaciones de requisitos entre cursos, arrastrando cada requisito sobre el curso correspondiente.

* Ph.D, prof. Esc. Ciencias de la Computación e Informática, Fac. Ingeniería, UCR
** Licda., prof. Esc. Ciencias de la Computación e Informática, Fac. Ingeniería, UCR.

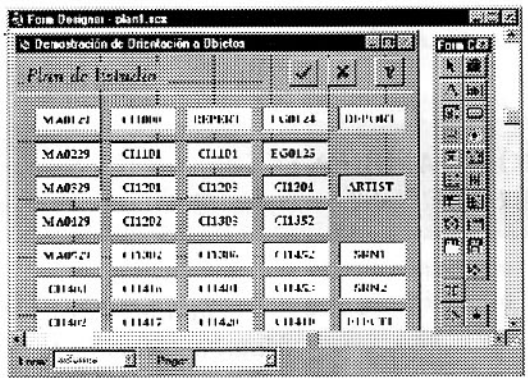


Figura N°1: Diseñador de formas de VFP

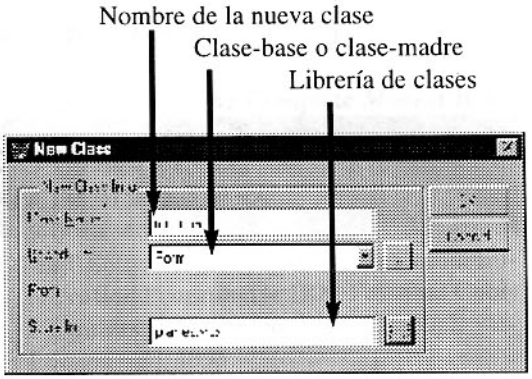


Figura N°2: Creación de una nueva clase

2. Características básicas de orientación a objetos en VFP

2.1. Clases

La aplicación en la cual se utiliza esta forma, posiblemente, está compuesta por varias ventanas con funcionalidades asociadas a la ventana del ejemplo. Estas ventanas deben ser uniformes en su interfaz, lo que implica la definición de una clase que las represente. Además, los distintos botones deben tener características similares (tamaño, color) dentro de cada forma y en relación con el resto de las formas (imagen, funcionalidad), por lo que debe crearse una jerarquía de clases de botones. Por último, la apariencia y el comportamiento de cada uno de los 37 cursos representados en el plan son iguales y se definen en una clase particular. Mediante la definición de clases, VFP agrupa estas propiedades y métodos en un solo módulo.

Para la creación de las clases del ejemplo, se utiliza el modelo de objetos de VFP, el cual incluye un conjunto de clases ya construidas, llamadas base, que se utilizan como punto de partida para crear nuevas clases. Además, se pueden crear nuevas clases basadas en otras que se hayan construido previamente. Todas éstas son guardadas por VFP en librerías de clases visuales.

Para crear una clase que estandarice las ventanas de la aplicación, se da la información en la ventana de creación de nuevas clases presentada en la figura N° 2.

En este caso, la nueva clase "miforma" hereda las propiedades y los métodos de la clase-base Form de VFP. Luego de proporcionar esta información, VFP activa su herramienta llamada diseñador de clases (Visual Class Designer) que permite diseñar y modificar visualmente las clases. En este caso se define el tamaño, el color, el título y el ícono de la forma, como se muestra en la figura N° 3.

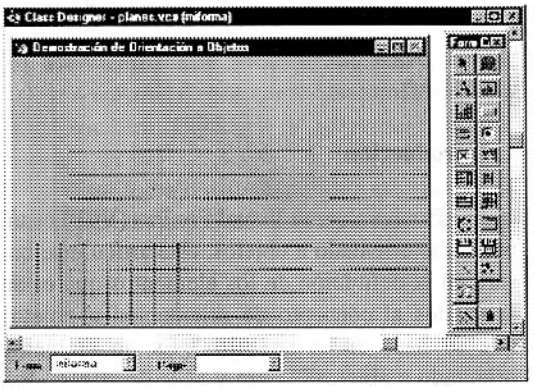


Figura 3: Diseñador de clases visuales de VFP

Las características definidas para esta nueva clase "miforma", que será utilizada para construir todas las formas de la aplicación, constituyen valores particulares para las propiedades heredadas de la clase Form, como se observa en el código asociado a la definición visual que se hizo de la clase, generado por VFP:

```
**************************************
*— Class:        miforma (d:\xx\planes.vcx)
*— ParentClass:  form
```

```
*— BaseClass:    form
*
DEFINE CLASS miforma AS form
      Height = 323
      Width = 423
      DoCreate = .T.
      ShowTips = .T.
      AutoCenter = .T.
      BackColor = RGB(192,192,192)
      Caption = "Demostración de Orientación a
                Objetos"
      Icon = "pez.ico"
      Name = "miforma"
ENDDEFINE
*
*— EndDefine: miforma
**************************************
```

Una vez creada la nueva clase, puede modificarse a conveniencia. Por ejemplo, una propiedad de una clase no está limitada simplemente a ser una variable, puede ser también un objeto de otra clase. En esta forma se podría haber incluido un botón común para todas las formas de la aplicación. También las propiedades y los métodos de una clase pueden protegerse. Si se decide proteger alguna propiedad o algún método, se podrá tener acceso a ellos sólo por otros métodos definidos en la misma clase. Si un objeto de otra clase necesita emplear una propiedad protegida o utilizar un método protegido, deberá hacerlo mediante mensajes a métodos que se hayan definido explícitamente para hacerlo.

Además del diseñador de clases, VFP provee un explorador de clases (Class Browser) que proporciona una forma gráfica y agradable de trabajar y manipular las librerías de clases, mostando las clases jerárquicamente o en orden alfabético con todos sus métodos, propiedades y objetos miembros. En la figura N° 4 se muestra el explorador de clases con todas las clases necesarias para el ejemplo que se analiza, las cuales se describirán más adelante.

2.2. Herencia

Por medio del explorador de clases y del diseñador de clases se crean las clases de botones. En primer lugar, se define una clase "botón" basada en la clase-base CommandButton, con las características generales (tamaño y color) de todos los botones que se utilizarán en la aplicación. Su código es el siguiente:

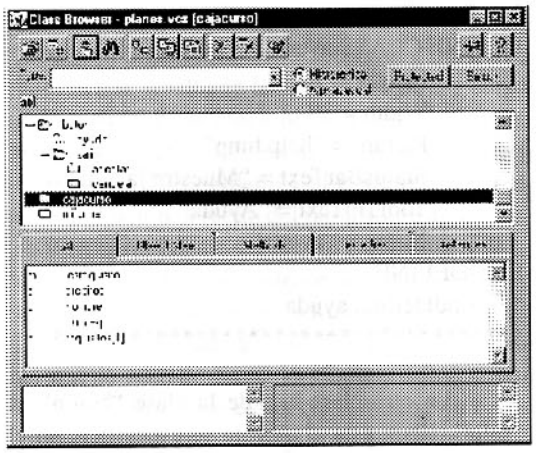


Figura N°4: Explorador de clases

```
**************************************
*— Class:        boton (d:\xx\planes.vcx)
*— ParentClass:  commandbutton
*— BaseClass:    commandbutton
*
DEFINE CLASS boton AS commandbutton
        Height = 30
        Width = 30
        Picture = ""
        Caption = ""
        Name = "boton"
ENDDEFINE
*
*— EndDefine: boton
**************************************
```

Con base en esta nueva clase "botón" se crean dos clases-hijas que, posiblemente, sean utilizadas en la mayoría de las formas de la aplicación: "ayuda" y "salir". En la clase del botón de ayuda se especifica su ícono particular y algunos mensajes que describen la función del botón:

```
**************************************
*— Class:        ayuda (d:\xx\planes.vcx)
*— ParentClass:  boton (d:\xx\planes.vcx)
```

```
*— BaseClass:   commandbutton
DEFINE CLASS ayuda AS boton
        Height = 30
        Width = 30
        Picture = "help.bmp"
        StatusBarText = "Muestra la ayuda"
        ToolTipText = "Ayuda"
        Name = "ayuda"
ENDDEFINE
*— EndDefine: ayuda
**************************************
```

La otra clase-hija de la clase "botón" es "salir". Para esta clase-hija se especifica, en el método Click, el comportamiento o tarea (salir de la ventana) que debe realizar este botón al ser presionado. Cabe destacar que este método Click pertenece a un tipo especial de métodos que provee VFP llamados "Eventos". Estos métodos se activan cuando el objeto detecta que ocurrió un evento particular. Por ejemplo, el método constructor (Init) y el destructor (Destroy) de la clase se ejecutan, automáticamente, cuando se crea un nuevo objeto y cuando se destruye, respectivamente. El código de la clase "salir" es el siguiente:

```
**************************************
*— Class:        salir (d:\xx\planes.vcx)
*— ParentClass:  boton (d:\xx\planes.vcx)
*— BaseClass:    commandbutton
*
DEFINE CLASS salir AS boton
        Height = 30
        Width = 30
        Name = "salir"

        PROCEDURE Click
                thisform.release
        ENDPROC
ENDDEFINE
*
*— EndDefine: salir
**************************************
```

Para la clase "salir" no se define un ícono particular ni los mensajes que describen al botón, ya que existen dos tipos diferentes de botones "salir": el de aceptar y el de cancelar. Para ello, se derivan de la clase "salir" las clases-hijas "aceptar" y "cancelar", las cuales comparten la función de salir de la ventana y la heredan en el método Click de la clase "salir". La diferencia entre estas dos clases es que un botón de aceptar debe guardar los cambios realizados por el usuario y luego debe salir de la ventana. Mientras, un botón de cancelar no guarda los cambios y le solicita una confirmación al usuario antes de salir.

Para cada una de estas clases se especifican las características de presentación. Conviene aclarar que el método Click de la clase "aceptar" debería modificarse para incluir las funciones de guardar los cambios. Sin embargo, dado que estas clases son un ejemplo y no están respaldadas por una aplicación con su respectiva base de datos, no se realiza esta modificación. El código de la clase "aceptar" es el siguiente:

```
**************************************
*— Class:        aceptar (d:\xx\planes.vcx)
*— ParentClass:  salir (d:\xx\planes.vcx)
*— BaseClass:    commandbutton
*
DEFINE CLASS aceptar AS salir
      Height = 30
      Width = 30
      Picture = "checkok.bmp"
      StatusBarText = "Acepta los cambios y sale"
      ToolTipText = "Aceptar"
      Name = "aceptar"
ENDDEFINE
*
*— EndDefine: aceptar
**************************************
```

2.3. Polimorfismo

Como se mencionó anteriormente, la clase del botón de "cancelar" modifica, parcialmente, el método Click heredado de la clase "salir". Para ello, se especifica el código adicional de la clase y luego se llama por medio de un mensaje al método Click de la clase-madre (en este caso "salir"), de la siguiente forma:

```
************************************
*— Class:        cancelar (d:\xx\planes.vcx)
*— ParentClass:  salir (d:\xx\planes.vcx)
*— BaseClass:    commandbutton
*
DEFINE CLASS cancelar AS salir
   Height = 30
   Width = 30
   Picture = "cancelx.bmp"
   StatusBarText = "Cancela los cambios y sale"
   ToolTipText = "Cancelar"
   Name = "cancelar"

   PROCEDURE Click
     resp=messagebox("Seperderán los cambios."+;
          chr(13)+"¿Desea continuar?",36,+;
          "Confirmación")
     if resp=6
          salir::click
     endif
   ENDPROC
ENDDEFINE
*
*— EndDefine: cancelar
************************************
```

En este caso se está aplicando lo que se conoce como polimorfismo de inclusión, ya que en la clase "salir" y en la clase "cancelar" (las cuales están relacionadas) se utiliza el mismo nombre (Click) para obtener un comportamiento similar cuando son presionados, aún cuando el resultado programado para cada acción es ligeramente distinto. El polimorfismo de operación sería posible definiendo el método Click de otra clase que no esté relacionada con las clases de botones. Sin embargo, en este ejemplo no se define este método para otras clases no relacionadas.

La última clase que hay que definir, siguiendo con el ejemplo, es la llamada "cajacurso", la cual representa a los cursos del plan de estudio. La definición de esta clase es un poco más compleja que la de las anteriores ya que se ven involucrados un mayor número de propiedades de la clase y debe especificarse el comportamiento de la clase ante diferentes situaciones y eventos. Cuando el usuario presiona el botón izquierdo del ratón sobre alguna caja que representa un curso, se resaltará la misma cambiándole su color y desplegando información básica del curso. También se resaltarán los requisitos de ese curso. Para realizar estas funciones se redefinen los métodos LostFocus y GotFocus heredados de la clase-base TextBox. Además, el usuario puede presionar el botón derecho del ratón sobre algún curso y arrastrar la caja sobre otro curso para indicar que el primero es requisito del segundo. Para esto se redefinen los métodos MouseDown y DragDrop heredados de la clase TextBox y se añade un nuevo método llamado "EsRequisito". Este método indica si un curso en particular ya es requisito de un curso dado y, si lo es, devuelve información sobre cómo localizarlo.

```
************************************
*— Class:        cajacurso (d:\xx\planes.vcx)
*— ParentClass:  textbox
*— BaseClass:    textbox
*
DEFINE CLASS cajacurso AS textbox
   FontName = "Book Antiqua"
   FontSize = 10
   Alignment = 2
   BackColor = RGB(255,255,255)
   BorderColor = RGB(0,0,0)
   ControlSource = "this.name"
   DragMode = 0
   Enabled = .T.
   ForeColor = RGB(0,0,0)
   Height = 26
   HideSelection = .T.
   InputMask = "XXXXXX"
   ReadOnly = .T.
   Width = 70
   DisabledForeColor = RGB(0,0,0)
   DisabledBackColor = RGB(255,255,255)
   SelectedForeColor = RGB(0,0,0)
   SelectedBackColor = RGB(0,255,0)
   *— Nombre del curso
   nombre = ""
   *— Numero de requisitos
   numreq = 0
   *— Creditos del curso
   creditos = 4
   Name = "cajacurso"
   DIMENSION requisitos[1]
```

```
*— Devuelve .T. si parámetro es req. del curso
PROCEDURE esrequisito
  PARAMETERS sigla
  i=1
  encontrado=0
  do while (i<=this.numreq) and (encontrado=0)
    if sigla=this.requisitos(i)
    encontrado=i
    else
    i=i+1
    endif
  enddo
  return encontrado
ENDPROC

PROCEDURE MouseDown
  LPARAMETERS nButton, nShift,
   nXCoord, nYCoord

  if nButton = 2
    this.drag(1)
  endif
ENDPROC

PROCEDURE LostFocus
  wait clear
  this.backcolor=rgb(255,255,255)
  for i=1 to this.numreq
    frase= 'thisform.'+this.requisitos(i)+;
    '.backcolor=rgb(255,255,255)'
    &frase
  endfor
ENDPROC

PROCEDURE GotFocus
  this.backcolor=rgb(0,255,0)
  wait window
    this.nombre+chr(13)+"Créditos:";
  +Str(this.creditos,2) nowait noclear
  for i=1 to this.numreq
    frase='thisform.'+this.requisitos(i)+;
    '.backcolor=rgb(128,255,128)'
    &frase
  endfor
ENDPROC

PROCEDURE DragDrop
  LPARAMETERS oSource, nXCoord,;
  nYCoord

  if (oSource.name <> this.name)
  && Si no es el mismo curso **
    posicion =
      this.esrequisito(oSource.name)
    if posicion = 0 && Si ese curso aun
    no es requisito, agreguelo **
    this.numreq = this.numreq+1
    dimension ;
    this.requisitos(this.numreq)
    this.requisitos(this.numreq)=oSource.name
    else  && Si ese curso ya es
      requisito, eliminelo **
    for i=posicion to this.numreq-1
      this.requisitos(i)=this.requisitos(i+1)
    endfor
    this.numreq = this.numreq-1
    if this.numreq<>0
      dimension this.requisitos(this.numreq)
    endif
    endif
  endif
 ENDPROC
ENDDEFINE
*
*— EndDefine: cajacurso
**************************************
```

Obsérvese que en esta clase se agregaron algunas propiedades adicionales (nombre, creditos, requisitos y numreq) que no fueron heredadas de la clase-base TextBox, además de haberse añadido el método EsRequisito.

Con la definición de todas estas clases, en la barra de herramientas de la ventana de diseño de formas (figura N°1) es posible mostrar las clases construidas e instanciarlas como objetos de la forma final. Así se puede construir, rápidamente y en forma consistente, la ventana mostrada en la figura N° 1.

Es importante añadir que todas las clases utilizadas en este ejemplo tienen un objeto visual asociado que las representa. Sin embargo, en VFP pueden construirse clases no visuales que no tienen ningún objeto gráfico relacionado. Para definir estas clases se procede de la misma forma, mediante la opción de VFP, para crear una

nueva clase. La diferencia es que al indicar la clase-base o clase-madre de la cual parte, se especifica la clase-base Custom, que es una clase que provee VFP precisamente para este fin. Luego, por medio del explorador de clases, se le podrán definir nuevos métodos y propiedades. Esta clase servirá, si es necesario, como clase-madre para derivar otras clases.

Finalmente, en relación con la herencia, VFP no permite herencia múltiple. Solamente se puede utilizar como clase-madre a una clase-base de VFP o a una clase definida previamente por el desarrollador de la aplicación.

3. CaracterIsticas asociadas a orientaciOn a objetos en VFP

3.1. Chequeo de tipos de variables

En VFP se pueden utilizar dos tipos de variables: las que son locales a cada procedimiento o función y las que son públicas, es decir, que son accesibles para toda la aplicación. En el caso de las variables locales no es necesario declararlas, simplemente se utilizan cuando son necesarias y sólo podrán ser utilizadas en el módulo en que se les asignó un valor. Para las variables públicas es necesario indicar, antes de usarlas, su condición de públicas pero sin necesidad de declarar su tipo. Por ejemplo, si se desea que la variable *contador* sea accesible en toda la aplicación, debe realizarse, antes de utilizarla, la siguiente declaración:

```
PUBLIC contador
```

En el momento en que se le asigna un valor a una variable, sea pública o local, ésta asume el tipo de ese valor. Por ejemplo,

```
miVar = 13
```

Luego de hacer esta asignación, *miVar* es de tipo numérico y solo podrá ser utilizada en operaciones o funciones que acepten valores numéricos. Su tipo puede comprobarse utilizando la función type que devuelve el tipo de la variable:

```
? type('miVar')
```

Como resultado de este comando en pantalla se desplegará una "N" indicando que es de tipo numérico. Sin embargo, a pesar de que la variable ya tiene asignado el tipo numérico, es posible asignarle otro valor de otro tipo a la misma variable sin que ello cause un error. Por ejemplo:

```
miVar = "Hola"
```

En este momento la variable *miVar* asume el tipo caracter, el cual se corrobora luego de ejecutar el siguiente comando:

```
? type('miVar')
```

el cual despliega en pantalla una "C" indicando que el tipo de la variable *miVar* es caracter.

En cuanto a variables que son objetos de una clase, es perfectamente legal en VFP asignar un valor de un tipo de datos diferente a una variable que representa un objeto.

Como puede verse en los ejemplos presentados, VFP realiza un chequeo de tipos dinámico, en donde en tiempo de ejecución cuando se hacen las asignaciones a las variables, las variables asumen el tipo del valor que se les está asignando.

3.2. Liga de métodos

En relación con la liga de métodos, VFP realiza liga de métodos dinámica, la cual se lleva a cabo en tiempo de ejecución cuando se conoce el tipo de la variable. El siguiente programa es un ejemplo que muestra cómo a la misma variable objeto se le asocian dos métodos diferentes con el mismo nombre despliega según el valor que posee en el momento en que se llama al método:

```
PUBLIC objeto
objeto = createobject('padre')
objeto.despliega  && Se despliega Soy el padre
do cambia
objeto.despliega  && Se despliega Soy el hijo

PROCEDURE cambia
objeto=createobject('hijo')
ENDPROC

DEFINE class padre as custom
procedure despliega
        wait window "Soy el padre"
endproc
ENDDEFINE
```

```
DEFINE class hijo as padre
procedure despliega
        wait window "Soy el hijo"
endproc
ENDDEFINE
```

Para demostrar que esta liga del método despliega se realiza en tiempo de ejecución, se presenta el mismo ejemplo con unos pequeños cambios, en donde en la clase hijo se define el método muestra en lugar de despliega. Sin embargo, en el programa principal se sigue llamando al método despliega de objeto que generará un error en tiempo de ejecución y no en tiempo de compilación:

```
PUBLIC objeto

objeto = createobject('padre')
objeto.despliega  && Se despliega Soy el padre
do cambia
objeto.despliega  && Genera un error

PROCEDURE cambia
    objeto=createobject('hijo')
ENDPROC

DEFINE class padre as custom
    procedure despliega
        wait window "Soy el padre"
    endproc
ENDDEFINE

DEFINE class hijo as custom
    procedure muestra
        wait window "Soy el hijo"
    endproc
ENDDEFINE
```

Si se ejecuta este programa se notará que sólo en el momento de la ejecución VFP detecta que el método despliega no corresponde a la variable objeto, luego de habérsele reasignado otro tipo de objeto.

3.3. Manejo de memoria

En cuanto al manejo de memoria, VFP recupera automáticamente la memoria asignada a los objetos locales y a las variables locales de los procedimientos y de las funciones una vez que éstos hayan finalizado. Sin embargo, la liberación de la memoria de los objetos y variables públicos o los utilizados en programas principales debe realizarse en forma manual; el programador debe liberar la memoria mediante la rutina release. Para recuperar la memoria de la variable contador declarada como pública en un ejemplo anterior, deber digitarse el siguiente comando:

```
RELEASE contador
```

Luego de ejecutarse este comando, la variable contador no podrá volver a utilizarse a menos que se vuelva a declarar como pública o que se utilice como local en algún procedimiento o función.

3.4. Uniformidad

Hasta aquí se han descrito la mayoría de las características de VFP relacionadas con orientación a objetos. A pesar de que un programador puede utilizar estas características para construir aplicaciones totalmente orientadas a objetos, VFP no es un lenguaje totalmente uniforme ya que es posible diseñar aplicaciones sin necesidad de utilizar clases y objetos, especialmente en lo que se refiere propiamente a programación de rutinas. Esto hace que VFP sea un lenguaje híbrido, en donde el programador si lo desea puede seguir una metodología de programación estructurada o incluso puede mezclar varias metodologías de desarrollo.

4. Facilidades complementarias del ambiente de VFP

4.1. Interfaz visual

VFP posee características que lo hacen fácil de utilizar y que ayudan en el buen uso de los objetos. Una de estas características, que es lo primero que llama la atención de este ambiente de desarrollo, es la palabra visual en el nombre. ¿Por qué se le llama visual? ¿Tiene características de lenguajes visuales?

Según Microsoft Corporation, se le agregó la palabra visual al nombre de FoxPro para seña-

lar que Visual FoxPro (VFP) es un producto con un enfoque más orientado a herramientas y menos orientado al lenguaje para el desarrollo de aplicaciones de bases de datos [Microsoft, 1997]. Además, Microsoft señala que este soporte de herramientas visuales tiene como objetivo facilitar la creación y reutilización de objetos. Sin embargo, algunos autores no coinciden con el uso de la palabra visual para este producto. Ricardo Baeza, por ejemplo, opina que Visual Basic y toda la familia visual de Microsoft no son, a pesar de sus nombres, lenguajes de programación visual. Son lenguajes textuales que utilizan un constructor de interfaz gráfico para hacer la programación de buenas interfaces más fácil para el programador [Baeza, 1997].

Con base en la definición más flexible que se da el artículo Lara y Murillo, 1997 acerca de un ambiente visual, se considera a VFP como un ambiente de desarrollo visual que provee herramientas que le permiten al programador observar la interfaz final mientras la construye. "Esta característica, al mostrar interactivamente lo que se construye, apoya la concepción o la visión orientada a objetos que debe poseer el programador para diseñar su aplicación" Lara y Murillo, 1997.

La interfaz de VFP totalmente gráfica facilita la creación de aplicaciones, las cuales requieren un mínimo de escritura de código pues la mayoría de las funciones se realizan a través de su interfaz. Por ejemplo, Visual FoxPro provee herramientas que facilitan las tareas avanzadas asociadas con la creación y el manejo de clases. El explorador de clases, presentado anteriormente, es un modelo que brinda toda la funcionalidad requerida y además es una arquitectura abierta que permite su adaptación por los programadores. El diseñador de clases también permite la creación y modificación visual de las clases.

4.2. Biblioteca de clases

La característica anterior está relacionada con la disponibilidad de la biblioteca de clases-base que brinda VFP, en la cual incorpora una diversidad de clases visuales para el diseño de interfaces y dos clases-base para derivar clases no visuales (Custom y Timer). VFP permite que el programador construya sus propias librerías de clases tanto visuales como no visuales.

4.3. Sintaxis

Otra de las características de VFP es la sencillez de la sintaxis de su lenguaje, la cual es en algún sentido natural en la medida en que se asemeja al idioma Inglés. Esto le facilita al programador el paso del diseño de la jerarquía de clases, de objetos y de su comportamiento a la programación en el lenguaje.

4.4. Depuración

Para la fase de prueba de programas, VFP posee dos herramientas para la depuración y prueba de programas. La herramienta "Trace" le permite al programador ejecutar un programa paso por paso, es decir, un comando a la vez. Esto le permite al desarrollador determinar el tipo de liga de métodos que VFP llevó a cabo durante la ejecución de la aplicación. Por otro lado, con la ventana del depurador de programas (Debug) el programador puede ver, mientras ejecuta el programa, el contenido en memoria de las variables u objetos que se están utilizando.

4.5. Tipo de ayuda

La ayuda en línea y sensible al contexto que provee VFP es un mecanismo que facilita la manipulación de los objetos y de todo el ambiente de programación. En el archivo de ayuda de VFP están documentados los objetos, las propiedades, los eventos y los métodos que están contenidos en el explorador de clases. Además, VFP hace uso de mensajes de error y de advertencia que mejoran la interacción entre el programador y el ambiente. La ayuda que brinda VFP es sobre todo descriptiva en cuanto a comandos y opciones del ambiente, realmente, no guía al programador en el aprovechamiento de la orientación a objetos del ambiente.

4.6. Herramientas

Adicionalmente, VFP posee un manejador de proyectos que facilita la integración y el ma-

nejo de todos los elementos de una aplicación (formas, reportes, bases de datos, programas, librerías de clases, etc.) y que permite la creación de archivos ejecutables que incluyen todos estos elementos.

VFP también posee un generador de documentación para proyectos y archivos de programas, el cual realiza un análisis sobre qué tan uniforme es el formato del código. Este generador tiene opciones para dar formato al código fuente, es decir, a las palabras reservadas del lenguaje, las variables definidas por el programador, indentación e inclusión de encabezados a diferentes partes del código. Además, genera algunos reportes tal como la jerarquía de relaciones que forman la aplicación.

5. Conclusiones

Luego del análisis realizado, la principal conclusión a la que se llega es que Visual FoxPro sí es un ambiente de desarrollo orientado a objetos, tanto para la definición de la interfaz como para la creación del código del procesamiento de la aplicación. Su intefaz, además de ser amigable, permite una construcción visual de los componentes de la aplicación y es orientada a objetos, ya que se manipulan elementos asociados a clases-bases de VFP.

En relación con las características básicas de orientación a objetos, VFP permite la abstracción de datos por medio de la definición de clases que agrupan en un solo módulo propiedades y métodos que describen a objetos similares. Sin embargo, la abstracción de datos realizada en VFP no es tan fuerte y puede violarse si el programador no declara las propiedades como protegidas, dejando abierta la posibilidad de que en cualquier momento una instrucción haga acceso directo a su contenido sin utilizar mensajes que invoquen al método respectivo.

En cuanto a la herencia, en VFP sólo es posible la herencia simple y no la múltiple. Sin embargo, es suficiente para que sea considerado como un ambiente orientado a objetos. La herencia múltiple es poco utilizada para la definición de clases en general, por lo que su ausencia en VFP no es crítica. Con la herencia, se da paso en VFP al polimorfismo de inclusión que permite que dos clases relacionadas posean métodos con el mismo nombre, ya sea que una clase sobreescriba el método de la otra clase o, simplemente, lo herede. Además, VFP permite el polimorfismo de operación entre dos clases no relacionadas. La decisión de si programarlo por sobrecarga o en forma paramétrica, depende del enfoque que le dé el programador al comportamiento asociado a los métodos.

Como consecuencia del polimorfismo surge la necesidad de entender aspectos tales como la liga de métodos. VFP realiza el chequeo de tipos y la liga de métodos en forma dinámica. Esto hace flexible al lenguaje ya que una misma variable u objeto podrá tener diferentes comportamientos, dependiendo de sus características en tiempo de ejecución. La responsabilidad que tiene cada objeto de llevar el control sobre su propio tipo, aumenta la definitiva orientación a objetos del lenguaje. Sin embargo, este dinamismo tiene tres desventajas. Primero: no es una buena práctica de programación asignarle a una misma variable valores de tipos diferentes. Esto hace poco legible el código y le resta lógica al programa. Segundo: a pesar de ser flexible, la ejecución es menos eficiente, ya que el chequeo y la liga consumen recursos en el momento de la ejecución. Tercero: este dinamismo aumenta la probabilidad de errores en tiempo de ejecución. Por esta razón, el programador debe ser prudente en el uso de esta flexibilidad.

Por otra parte, el manejo que hace VFP de la memoria es eficiente en cuanto a tiempo de ejecución, ya que al no tener un recolector de basura automático (excepto para las variables locales) no emplea recursos en tiempo de ejecución. Esto implica que el programador debe tener el cuidado de liberar memoria cuando no la ocupe para no agotar el recurso de memoria del sistema.

En cuanto a la uniformidad del lenguaje de VFP, por ser un lenguaje híbrido es posible obviar la programación por objetos, en casos triviales que no ameriten toda la formalidad de esta metodología de programación. Esto es apropiado cuando éste es el propósito buscado, luego de haber comparado los beneficios con el esfuerzo asociado de la programación orientada a obje-

CARACTERISTICA	DESCRIPCION
Encapsulación	Sí. • Por medio de las clases que encapsulan propiedades y métodos en una misma entidad.
Abstracción de datos	Sí. • Por medio de las clases • Dos niveles de visibilidad para las propiedades y los métodos: públicos y protegidos. • El acceso a los objetos se realiza por medio de mensajes y métodos.
Herencia	Sí. • Herencia simple. • El árbol de herencia se inicia con las clases-base de Visual FoxPro.

Polimorfismo	Sí.
	• De inclusión por medio de herencia.
	• De operación por medio de:
	  • Sobrecarga
	  • Paramétrico

Chequeo de tipos	• Dinámico.
Liga de métodos	• Dinámica.
Manejo de memoria	• Automática para variables locales. • Manual para variables públicas y de programas principales.
Uniformidad del lenguaje	• Híbrido. • Apoya el uso de la programación orientada a objetos pero no fuerza el uso de este paradigma.
Sintaxis del lenguaje	• Simple.

Ambiente de programación	• Amigable.
	• Integra:
	  • Construcción visual.
	  • Librería de clases.
	  • Explorador de clases.
	  • Diseñador de clases.
	  • Ayuda en línea sensible al contexto.
	  • Depuración de programas: herramientas "Debug" y "Trace".
	  • Administrador de proyectos.
	  • Generador de documentación.

Figura 5: Características de orientación a objetos de Visual FoxPro

tos. En este caso es valiosa la flexibilidad del lenguaje. En el caso contrario (en el que la no utilización de la programación orientada a objetos no es una decisión técnica, sino que generalmente es producto de utilizar otras metodologías de programación propias de otros lenguajes o de otras versiones de FoxPro), se estarán desaprovechando las ventajas y facilidades que ofrece el ambiente y que son propias de la programación orientada a objetos.

Finalmente, el ambiente general de VFP es muy amigable ya que, además de ser visual, provee herramientas gráficas que facilitan y hacen más agradable la interacción con el ambiente de desarrollo y de programación.

Como conclusión general puede afirmarse con seguridad que VFP es un ambiente orientado a objetos con todas las características que implica el concepto y, tomando en cuenta los lenguajes de programación predecesores de los cuales se ha derivado la creación de VFP, se ha demostrado que es un ambiente de programación que evoluciona. En este sentido es de esperar que VFP en el futuro incorpore una librería de clases no visuales más amplia que la actual, la cual incluye en este momento las clases Custom y Timer. Además, se esperaría que consolide su enfoque de orientación a objetos a nivel de administración de base de datos.

Adicionalmente, es interesante destacar que la programación orientada a objetos está relacionada, en alguna medida, con la programación por eventos. VFP podría ser analizada desde este otro paradigma, al poseer características tales como métodos que responden a eventos dados en el sistema.

En la figura N° 5 se resumen los resultados expuestos en este artículo.

6. Bibliografía

Almunia Sanz, Pablo. *Introducción a la POO en Visual FoxPro (I).* 1995. Fuente: http://www.ourworld.compuserve.com-/homepages/palmun/vfp.htm

Almunia Sanz, Pablo. *Introducción a la POO en Visual FoxPro (II).* 1995. Fuente: http://www.ourworld.compuserve.com-/homepages/palmun/vfp.htm

Griver, Alan Y. The Visual FoxPro 3 Codebook, *Creating Reusable Classes with Visual FoxPro.* 1995. Fuente: http://www.microsoft.com/vfoxpro/vfp_ood.htm

Griver, Alan Y. The Visual FoxPro 3 Codebook, *The Visual FoxPro Object Model.* 1995. Fuente: http://www.microsoft.com/vfoxpro/vfp_ood.htm

Lara, Vladimir y Murillo, Maureen. *¿Es un ambiente orientado a objetos?.* Revista de Ingeniería. Volumen 6, Número 2. San José, Costa Rica: Editorial de la Universidad de Costa Rica. 1997.

Microsoft Corporation. *Explanation and Examples of Non-Visual Classes.* Estados Unidos: Microsoft Developer Network. 1995. Fuente: http://www.microsoft.com/vfoxpro/vfp_ood.htm

Microsoft Corporation. *Managing Classes with Visual FoxPro.* Estados Unidos: Microsoft Developer Network. 1995. Fuente: http://www.microsoft.com/vfoxpro/vfp_ood.htm

Microsoft Corporation. *Visual FoxPro 5.0 Guided Tour.* 1997. Estados Unidos. Fuente: http://www.microsoft.com/vfoxpro