26
24

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFECIONALES ACATLAN
E. N. E. P. ACATLAN

" PROGRAMACION ORIENTADA A OBJETOS: SU APLICACION UTILIZANDO CLIPPER 5"

T E S I S

Que para obtener el Título de:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P r e s e n t a:

SILVIA VEGA OBREGON

Santa Cruz Acatlán, Méx. 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DEDICATORIAS

Mi tesis:

Hoy ya no es tan solo un anhelo concebido años atrás, sino una gran realidad; que es significado de muchos esfuerzos compartidos con varias personas, algunas de ellas hoy cercanas, otras lejanas, pero todas muy valiosas para mí, que a lo largo del camino han ido colaborando conmigo para poder conseguirlo, y a los cuales agradeceré infinitamente sus importantes aportaciones, que han sido guía y soporte para llegar a la meta propuesta: ***MI CARRERA PROFESIONAL.***

Es por éste motivo que dedico la presente:

A DIOS porque en tí señor he encontrado siempre la fuerza necesaria para poder seguir adelante y porque contigo cerca de mí cada paso ha sido siempre una meta.

A MIS PADRES con todo mi amor y respeto por ser las personas más importantes en mi vida, los cuales con su amor, sus palabras de aliento y sus inagotables fuerzas han puesto frente a mí un ejemplo de lucha constante, para conseguir la superación personal

A MIS HERMANOS con mucho cariño por estar siempre a mi lado cuando los necesito.

A ENRIQUE mi hermano con cariño y admiración por sus palabras de aliento y por su ayuda incondicional.

A AURELIO con amor por acompañarme y ayudarme durante toda mi carrera profesional y por motivarme para seguir siempre adelante.

A DAVID con cariño por darme su amistad y por las facilidades, disposición y ayuda desinteresada que siempre me brindo en todo momento, apoyando el desarrollo de mi trabajo de tesis.

A ERNESTO con cariño por darme su amistad, su compañía y sus consejos que siempre me ayudaron para seguir adelante.

A JORGE ARTURO quien asesoró y revisó mi trabajo, dando generosamente su amistad, tiempo y experiencia.

A MIS SINODALES quienes revisaron mi trabajo, dándome sus valiosos consejos y sugerencias para el mejoramiento de este trabajo.

A TODOS MIS PROFESORES por sus conocimientos y consejos transmitidos,y porque hicieron más grato mi aprendizaje gracias a su calidad humana.

Y A TODA LAS PERSONAS que esté omitiendo, pero que me han apoyado siempre que la he necesitado.

A todos ustedes por siempre...

GRACIAS

S.V.O.

REFLEXIONES

Detente en el umbral de la vida, que es la juventud, para preguntarte que te conviene ser, y empeña todo tu interés y todo tu esfuerzo en prepararte para una vida útil. Estudia, con interés empeño y alegría, para que tu actividad resulte siempre fecunda y sea un impulso hacía nuevas empresas. Para ello es preciso que te inspires en la idea de que cuanto haces hoy, es como la semilla que ha de dar frutos en un mañana próximo.

Encariñate con tu obra, por modesta que sea; el amor por lo que haces te ayudará a vencer las dificultades que por fuerza hallarás en todos tus caminos, te alentará en todos tus momentos; te compensará de todos tus esfuerzos. Los más grandes entre los buenos comenzaran siendo pequeños; pero, un día y otro día, multiplicaron sus afanes, renovaron sus propósitos, y, trabajando siempre con la alegría de la esperanza, alcanzaron cada vez una nueva meta. Sé tú así sonriendo, estudia, persiste, realiza, para que "cada uno de tus pasos sea una meta, sin que deje de ser un paso".

INDICE

INTRODUCCIÓN

Ante la velocidad de desarrollo de la tecnología, y el hecho de vivir en la era de la información se requiere de software que optimice la manera de encontrar, asimilar y actuar sobre la información requerida.

Una alternativa para la obtención de este software es la Programación Orientada a Objetos que es una técnica de construcción de aplicaciones, que funciona similar a como lo hace la mente humana, agrupando las propiedades más características de cada objeto.

El lenguaje manejador de bases de datos, Clipper 5, aunque no sea un lenguaje 100% orientado a objetos, es un lenguaje altamente estructurado con posibilidad de orientación a objetos, además de que es el mas utilizado en el desarrollo de sistemas de información y permite que el profesionista experto en este lenguaje no tenga que aprender otro para poder obtener los beneficios de la programación orientada a objetos.

El presente trabajo tiene la finalidad de dar a conocer los fundamentos, aplicación y uso de esta metodología de diseño y programación de sistemas de información. A lo largo del trabajo se avanzará desde el nivel más elemental, adentrándonos paulatinamente en el nivel más elevado de esta metodología, terminando con el desarrollo de un sistema de información

aplicando todos los conceptos analizados, que servirá de base para posteriores investigaciones a los Licenciados de Matemáticas Aplicadas y Computación, así como a las carreras afines.

El primer capítulo da a conocer la problemática actual del desarrollo del software, así como un breve panorama de la Tecnología Orientada a Objetos, su historia, su impacto actual, su futuro, sus beneficios y los lenguajes más comunes.

En el segundo capítulo se analizan los conceptos básicos de la Programación Orientada a Objetos, que fueron desarrollados a raíz del lenguaje SIMULA, como son: objeto, mensaje, método, clase, subclase y superclase, además de conocer la filosofía que encierra esta metodología, a través del llamado triángulo orientado a objetos. Se analizará conceptos tales como: encapsulación, polimorfismo, herencia, interacciones con objetos, agregación y asociación.

En el tercer capítulo se recordará brevemente la metodología tradicional de desarrollo de sistemas información y se dará a conocer las metodologías de desarrollo de sistemas orientadas a objetos, analizando la más conocida: la de Booch.

En el cuarto capítulo se conocerán las clases que vienen definidas en Clipper para crear objetos, tales como: la clase Error, la clase Get, la clase TBrowse, la clase TBColumn. Así como las referencias de los comandos y funciones de Clipper 5.2 utilizados en la programación del sistema.

En el quinto capítulo se aplicará esta metodología a un sistema sencillo, utilizando el método de Booch para realizar el análisis, diseño y programación del sistema utilizando Clipper 5.2

Los resultados del método de estudio realizado, nos arroja una sistematización de información de diversas referencias bibliográficas, así como el diseño de un programa con sus aspectos relevantes de cobertura aplicables en cualquier entidad social. Resulta impredecible, cual es la labor que se dará en la vida profesional, pero la innovación la actualización y el espíritu para el "cambio" son los pilares que deben prevalecer en cualquier disciplina.

I ANTECEDENTES HISTÓRICOS

1.1 TECNOLOGÍA ORIENTADA A OBJETOS

Comenzaremos este análisis con una breve descripción de lo que es la Tecnología Orientada a Objetos, así como una breve semblanza de la historia de la programación de Computadoras. También trataremos los antecedentes históricos más importantes de la Tecnología Orientada a Objetos, desde sus orígenes hasta nuestros días, describiendo brevemente los lenguajes de Programación Orientada a Objetos más significativos en la actualidad.

Para una mejor comprensión de la idea general de este capítulo, repasaremos brevemente algunos conceptos básicos de Programación. Describiremos primero lo que es un programa, en el campo de la Informática.

PROGRAMA

Es una serie de instrucciones que dicen a una computadora como llevar a cabo acciones específicas. Hay diferentes tipos de programas de computadora, desde los programas más elementales, para realizar cálculos sencillos o accesar datos, hasta programas más complejos con características de almacenamiento y recuperación de información o cálculos

complejos, como por ejemplo, sistemas de contabilidad, de control de inventarios, de control de producción, etc.

SISTEMA DE INFORMACIÓN

Es el conjunto de procedimientos y dispositivos implicados en el proceso, almacenamiento y distribución de la información en una organización.

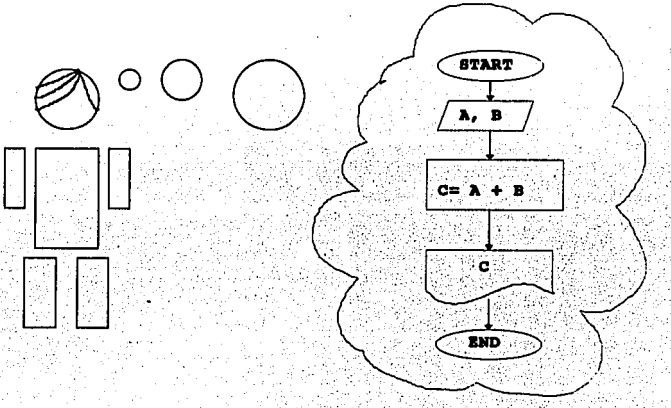


Figura 1

Programa para un cálculo sencillo, desarrollado por un programador.

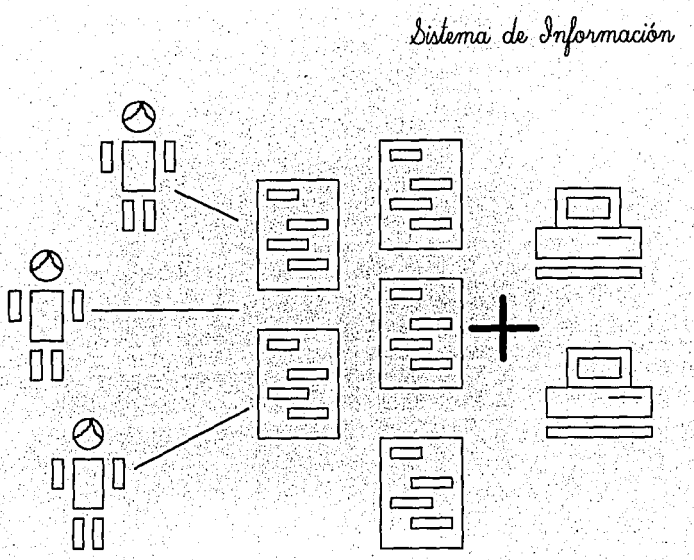


Fig. 2
Sistema de información. Desarrollado por varios programadores.

Los programas para computadora pueden ser construidos con un solo procedimiento, o secuencia de instrucciones que ejecutan la tarea deseada. Un procedimiento puede ser escrito por un solo programador, quien tiene la imagen mental del procedimiento entero. Sin embargo, los programas o sistemas más

grandes no pueden ser construidos de esta forma, es decir, por un solo programador, debido a la magnitud de éstos. La solución a este problema es utilizar la programación modular.

PROGRAMACIÓN MODULAR

Consiste en dividir el programa grande en pequeños componentes (procedimientos) que pueden ser construidos independientemente, luego combinar éstos hasta formar el sistema completo. "Bajo este principio se ha venido construyendo Software en los últimos 40 años".[1]

A un procedimiento se le denomina subrutina, que es el elemento más importante de la programación modular, que se inventó por los años 50's y es creada por una secuencia de instrucciones fuera de la rutina principal y dándole un nombre diferente, una vez definida puede ser ejecutada simplemente incluyendo este nombre en el programa que lo requiera. "Las subrutinas proveen el mecanismo básico de la programación modular, sin embargo, se necesita mucha disciplina para crear software bien estructurado. Sin esta disciplina es muy fácil escribir programas complicados que son resistentes a cambios, difíciles de entender e imposibles de mantener".[2]

A finales de los 60's, los científicos de la computación, hicieron un esfuerzo por desarrollar esta metodología considerandola un estilo para la programación.

---

(1) David A.Taylor
Object-Oriented Tecnology: A manager guide
Editorial Adison-Wesley
Estados Unidos, 1992. Página 3

(2) Ibid. Página 4

El resultado fue el refinamiento de la Programación Modular dentro de una tecnología conocida como Programación Estructurada.

PROGRAMACIÓN ESTRUCTURADA

La Programación Estructurada se apoya en la "descomposición funcional, en la cual un programa es sistemáticamente dividido en componentes, los cuales a su vez se subdividen en subcomponentes y así, hasta el nivel individual de subrutina".[3]

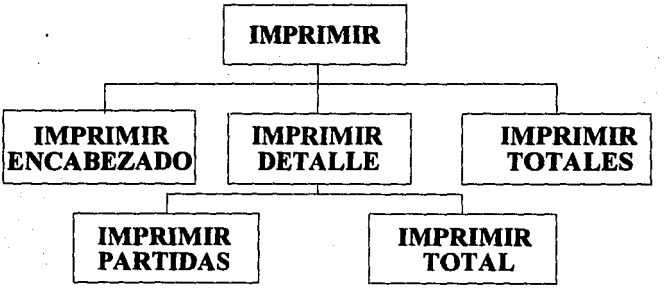


Figura 3.
Esquema de la Descomposición Funcional de un problema.

[3] Ibid. Página 4

Equipos separados de programadores escriben los diversos componentes, los cuales una vez probados, son ensamblados dentro del programa completo. También la Programación Estructurada se caracteriza por:

- Utilizar estructuras de control bien definidas como son: las selectivas y las repetitivas.
- Crear programas con un solo punto de entrada y un solo punto de salida.

La Programación Estructurada ha producido significantes mejoras en la calidad del Software en los últimos 20 años, pero "sus limitaciones son ahora aparentes. Uno de los más serios problemas es que es imposible anticipar el diseño de un sistema completo antes de que sea implementado".[4]

Después de 40 años de la invención de la subrutina, todavía se construyen sistemas a mano, es decir, una instrucción a la vez. Uno de los métodos para mejorar a la Programación Estructurada es la Programación Orientada a Objetos, la cual "es capaz de manejar tanto sistemas grandes, como pequeños, además de crear sistemas confiables que sean flexibles, sostenibles y capaces de desarrollar los cambios necesitados".[5]

"Son cada vez más quienes recurren a la Programación Orientada a Objetos para dinamizar el proceso de la creación de código reutilizable a partir de bibliotecas muy diversas. En este momento los métodos tradicionales de programar están tambaleándose".[6]

---

(4) Ibid. Página 5
(5) Ibid. Página 14
(6) Ann Steffora
Revista Personal Computing México.
Editorial Servicios Editoriales Sayrols, S.A. de C.V.
Agosto de 1993. Página 44

La Programación Orientada a Objetos ha causado, en ciertos círculos, una "verdadera revolución en cuanto a la manera de codificar. No sólo porque el cambio de paradigma de programación implica diferencias conceptuales, de organización y de ejecución, sino porque el ambiente en que se trabaja debe necesariamente cambiar para hacerse más productivo".[7]

Aunque la Programación Orientada a Objetos ha salido a la luz recientemente, tiene más de 20 años de existencia. Los conceptos básicos de la Tecnología Orientada a Objetos fueron introducidos en el lenguaje de programación SIMULA, desarrollado en Noruega a finales de los 60's. Tenemos aquí una interrogante interesante; Si la Programación Orientada a Objetos tiene ya varios años, entonces ¿Por qué durante tanto tiempo se había ignorado?. La respuesta tiene que ver con la complejidad del Software que actualmente se escribe. Los programas fuente se han vuelto más largos y extensos y por lo tanto más difíciles de mantener y modificar.

Hasta hace poco la Orientación a Objetos era sólo estudio de algunos académicos y no tenía relevancia práctica debido a los enormes requisitos de procesamiento. Quienes intentaron aplicar esta tecnología hace pocos años se encontraron con grandes problemas, porque los requisitos excedían las capacidades típicas del Hardware de aquella época. Ahora el Hardware ya ha mejorado su velocidad.

---

[7] Abel Archundia Pineda y Jorge Muñoz Márquez
Revista PC/TIPS BYTE México # 52.
Mayo de 1992. Página 92

Veamos ahora algunos beneficios de esta Tecnología. La base de la Programación Orientada a Objetos es la Herencia. "Un entorno orientado a objetos puede crear unidades de código reutilizable y encapsulado que sean extensibles. En tiempo de ejecución cada unidad de código u objeto debe heredar dinámicamente el conjunto de atributos que lo hacen único, sin tener que recompilar o enlazar de nuevo".[8]

La Programación Orientada a Objetos cumple con tres propósitos fundamentales:

1.- "Permite armar aplicaciones que pueden modificarse rápidamente y, una vez desarrolladas, las puede utilizar prácticamente cualquier persona.
2.- Los segmentos de código vienen encapsulados, por lo cual se libera al desarrollador de la necesidad de dominar una gran cantidad de conocimientos.
3.- Promueve la reutilización".[9]

La Programación Orientada a Objetos permite a los programadores pensar en términos que modelan de una manera más exacta el mundo real. "En lugar de escribir procedimientos que manipulen información como una entidad separada, la Programación Orientada a Objetos permite definir los objetos de diferentes tipos que combinan atributos (datos) y comportamiento (procedimientos) en un simple paquete".[10]

---

(8) David Baum
Revista PC WORD (España) Número 83, Diciembre 1992
Artículo: La Evolución del desarrollo de aplicaciones
Editorial IDG Communications. Página 226

(9) Ann Steffora Ob.Cit. Página 44

(10) Bruce D. Schatzman
Revista PC/TIPS BYTE México # 58
Noviembre de 1992. Página 88

Recientemente se ha escuchado acerca de Sistemas Operativos Orientados a Objetos, que tienen la característica de ser independientes del hardware en el que operan, es decir, se supone que no están ligados a ninguna plataforma de hardware específica. La tecnología capaz de enfrentarse con este problema (la interoperatividad) es; la Programación Orientada a Objetos. En un futuro no muy lejano la Programación Orientada a Objetos sustituirá a la programación tradicional orientada a procedimientos. La diferencia básica entre estos dos paradigmas de programación es que la Programación Orientada a Objetos utiliza mensajes entre objetos, en lugar de llamadas a subrutinas y opera con una organización fuertemente jerarquizada.

Resumiendo lo anterior podemos sacar como conclusión que la Programación Orientada a Objetos tiene tres beneficios principales:

- Agrega Calidad al Software
- Permite el Rápido Desarrollo
- Crea Código Reutilizable

Además este tipo de programación, es una nueva y abstracta forma de pensar en la forma de dividir los problemas. Esto implica un cambio de mentalidad acerca de la forma de resolver problemas por medio de una computadora, además de toda una serie de conceptos nuevos, algunos similares, otros inaplicables en la Programación Estructurada. Sin embargo, este cambio trae consigo beneficios implícitos, como lo son; flexibilidad, modularidad y reusabilidad de código para futuras aplicaciones. La suposición principal de esta tecnología es pensar en los problemas (que se puedan resolver por una computadora) como objetos del mundo real. Con este breve panorama de lo que es la Tecnología Orientada a Objetos, sus

antecedentes históricos y su finalidad, revisaremos en la siguiente sección la evolución histórica de los más importantes lenguajes de programación que utilizan esta metodología de desarrollo, junto con una breve descripción de la historia general de los lenguajes de programación en general y de los paradigmas de Programación más importantes de la actualidad.

1.2 PARADIGMAS DE PROGRAMACIÓN

Dentro del área de la programación de computadoras, existe el hábito de acostumbrarse a cierto lenguaje o metodología de programación. No obstante, el programador operando bajo un paradigma nuevo o diferente experimenta una alteración en su conceptualización del proceso de resolución de problemas. Hay una modificación esencial de perspectiva sobre lo que es un problema y lo que representa una solución.

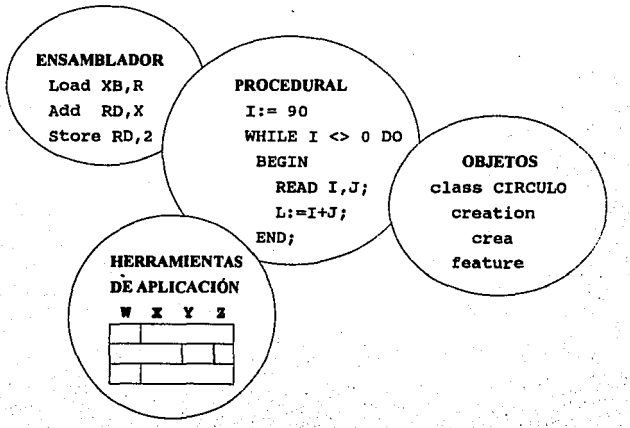


Figura 4
Paradigmas de programación. Fuente: Revista Soluciones Avanzadas # 5.

En la programación de computadoras, ha habido diversas formas de pensar, es decir, formas de concebir un problema y programar (o resolver) su solución. Los paradigmas de programación más importantes son los siguientes:

1.- Lenguaje de máquina y ensamblador
2.- Procedural
3.- Orientado a Objetos
4.- Herramientas para aplicaciones.

La Programación en Ensamblador es la primera disciplina de programación de computadoras digitales. A continuación se muestra una rutina en macroensamblador que lee el contenido de un archivo en disco.

```
READF PROC FAR
;
;       . DESCRIPCION: Esta rutina lee un archivo en
;         en disco.
;
      PUSH DS
                          ;
      MOV AX, SEG DATARD
      MOV DS, AX
      MOV DI, 0           ; Inicializa apuntador
                          ; Lee archivo
DO1:
           MOV SI, 0      ; Inicializa apuntador
           MOV AH, 14H    ; Inicializa lectura
           LEA DX, FCBLB1
           INT 21H
                           ;
           CMP AL,0        ; Verifica si se efectuó lectura
           JE IF1
                           ;
              CMP AL,1     ; EOF
              JE OUT1      ; Salto para procesar entrada al
                           ; disco
              MESSG 9      ; DTA muy pequeña
              POP DS
              RET
                              ;
```

```
                                    ;
IF1:        MOV AL,DDTA1 [SI]       ; Carga primer byte del
                                    ; área DTA
            MOV AH,DDTA1 [SI+1]     ; Carga segundo byte del
                                    ; área DTA
            MOV DDATA [DI],AX       ; Transfiere palabra DTA
                                    ; a memoria
            ADD DI,2                ; Incrementa longitud del
                                    ; índice de palabra
      JMP DO1
                                    ;
OUT1:
      MOV COUNT, DI                 ; Salva cuenta
      MOV DX,0                      ; Impresora 0
      MOV AH,1                      ; Inicializa impresora
      INT 17H
                                    ;
      MOV DI,0                      ; Apuntador a datos del disco
READF ENDP
```

El paradigma de programación más duradero ha sido el de la programación procedural, que data de la creación del lenguaje FORTRAN, en 1957, por de J. Backus.

La Programación Estructurada o Paradigma Procedural es el más usado por todos los programadores de computadoras digitales. En este paradigma, se visualiza una solución de un problema en términos de una jerarquía de procedimientos para la manipulación de datos. Dentro del paradigma procedural, "los datos como entidades, juegan un papel subordinado al de los procedimientos, son receptores pasivos de las manipulaciones. La organización de un sistema computacional está inicialmente pensada, después implantada y finalmente descrita en términos de procedimientos".[11]

---

[11] Warren R. Greiff
Revista Soluciones Avanzadas
Número 5, Septiembre-Octubre 1993
Editorial Xview, S.A. de C.V. Página 36

A continuación se muestra una subrutina en Turbo Basic que lee el contenido de un archivo en disco.

```
SUB LECTURA
    OPEN "ARCHIVO.DAT" FOR INPUT AS #1
    CLS
    PRINT "NUMERO"
    PRINT "------"
    DO WHILE NOT EOF(1)
       INPUT #1, NUMERO
       PRINT NUMERO
    LOOP
    PRINT "PULSE UNA TECLA PARA CONTINUAR"
    X$=INPUT$(1)
    CLOSE #1
END SUB
```

En el paradigma Orientado a Objetos, que es el más reciente, los objetos se comunican entre sí, por medio de mensajes, pidiendo unos a otros que efectúen tareas que a ellos les corresponde o que provean información sobre su estado actual. Las acciones son definidas como métodos de objetos que responden al paso de mensajes en vez de procedimientos invocados por llamadas. "Ha generado una concepción distinta de la programación, dejando de ser jerárquica y convirtiéndose en heterárquica o reticular (de módulos independientes)".[12]

A continuación se muestra un trozo de programa en Clipper que lee el contenido de un archivo en disco utilizando objetos.

```
obj:=TBrowseDB(Arriba,Izquierda,Abajo,Derecha)
obj:HeadSep := "=+="
obj:ColSep:= "|"
SELECT 1
SEEK Fdbf
```

[12] Ibid. Página 43

```
//Carga las columnas del Browse
WHILE Fdbf == ALLTRIM(TABLAS->ARCHIVO)
      columna:= TBColumnNew(ALLTRIM(TABLAS->NUMEROS),;
      Y:=1
      Arreglo[++X,Y++]:=TABLAS->NUMEROS
IF TABLAS->TIPO <> "O"
   obj:addColumn(columna)
ENDDO
```

El desarrollo de software de Herramientas para aplicaciones estuvo durante mucho tiempo muy limitado. Estas herramientas de desarrollo de aplicaciones son los conocidos "paquetes", las características comunes a todos ellos son la forma de efectuar la entrada de datos y la selección de funciones por medio de menús o formatos predefinidos.

La utilización de los diferentes paradigmas está bien definida; por ejemplo, el lenguaje ensamblador lo utilizan programadores de sistemas (los programas son muy elaborados) y la Programación Estructurada y la Programación Orientada a Objetos son utilizados por programadores de aplicaciones (programas menos elaborados) y las herramientas de aplicación las utilizan mayoritariamente el resto de los usuarios (no necesariamente programadores).

1.3 LENGUAJES DE PROGRAMACIÓN ORIENTADA A OBJETOS

Antes de revisar cuales son los lenguajes de Programación más importantes que aplican la Programación Orientada a Objetos, repasaremos brevemente la historia de los lenguajes de Programación en general.

Actualmente se habla ya de generaciones en los lenguajes de programación, al igual que en el hardware. Sin embargo, no se comenzó a pensar en las diferentes generaciones de los lenguajes de desarrollo de aplicaciones hasta que surgió el término "lenguaje de cuarta generación" (4GL).

La Primera Generación de los lenguajes de desarrollo de Software se refiere al código o lenguaje máquina, que es el lenguaje básico binario (0's y 1's) que el procesador lee e interpreta para llevar a cabo sus operaciones.

La Segunda Generación de lenguajes de programación es el código Ensamblador o lenguaje Ensamblador. El lenguaje Ensamblador es un lenguaje de bajo nivel en el cual cada instrucción del programa corresponde a una o más instrucciones que el procesador puede realizar, las cuales se traducen a instrucciones de código máquina (0's y 1's). Los lenguajes ensambladores son dependientes del tipo de computadora asociado, ya que están diseñados para las características específicas y el conjunto de instrucciones de un determinado procesador. Su uso es básicamente para programar aplicaciones tales como, sistemas operativos o lenguajes de programación, pero no para el desarrollo de aplicaciones comerciales.

La Tercera Generación de lenguajes de programación son los llamados lenguajes de alto nivel, tales como, Pascal, COBOL, BASIC o C, los cuales utilizan traductores llamados compiladores e interpretes. Cabe señalar que la diferencia entre estos dos tipos de traductores, radica en que el primero traduce todo el programa a lenguaje máquina, para posteriormente ejecutarlo, en tanto que en el interprete traduce las intrucciones y las ejecuta una por una. Los lenguajes de alto nivel son procedurales es decir, el programador tiene que especificar un procedimiento que el computador debe seguir para realizar una tarea determinada. Se utilizan principalmente para programar aplicaciones comerciales y científicas. Algunos de ellos, como el lenguaje C, también se utilizan para programación e sistemas. (Sistemas Operativos, compiladores, hojas de cálculo, etc.)

A principios de los 80's surgió el término de Lenguaje de Cuarta Generación para describir una categoría de herramientas de programación interactiva usadas en el desarrollo de aplicaciones para sistemas de Administración de Bases de Datos. Hoy en día estos lenguajes abarcan entre otras aplicaciones; lenguajes de consulta (Query) para usuario final, generadores de informes, generadores de pantallas de entrada de datos y entornos completos de desarrollo, entre otros.

En esta cuarta generación de lenguajes de programación, se agregan también otro tipo de lenguajes de programación. Los lenguajes de Programación Orientada a Objetos, los cuales son lenguajes de programación no procedural en los cuales los elementos del programa están considerados como Objetos que pueden pasar mensajes a otros objetos. En un Programa Orientado a Objetos, cada objeto tiene su propio código de datos y su propio código de programa, y es independiente por si mismo.

Dentro de los lenguajes de Programación Orientados a Objetos se pueden distinguir dos ambientes de trabajo los "puros" e "híbridos". Los híbridos ofrecen la orientación a objetos como un añadido a un lenguaje tradicional. En cambio, en un lenguaje puro, se construye con la Tecnología Orientada a Objetos desde su diseño, incluyendo características como herencia, polimorfismo y encapsulación, los cuales se analizarán más adelante.

La programación en un ambiente híbrido es semejante a programar en un lenguaje tradicional, por lo que resulta muy fácil el salto de un lenguaje tradicional a uno en ambiente de objetos de tipo híbrido, mientras que en los ambientes puros se requiere de un tiempo y esfuerzo mayor, pues es un ambiente nuevo de programación, para el usuario que ha programado gran parte de su vida en lenguajes tradicionales basados en funciones y procedimientos, ya que en la mayoría de los lenguajes puros utilizan interfases gráficas de usuario ó GUI.

El lenguaje ALGOL-60 dado a conocer al inicio de los 60's, fue el precursor del desarrollo de la Programación Estructurada. El precursor de los lenguajes de Programación Orientada a Objetos es SIMULA-67, nacido en Noruega, el cual "buscaba modificar a los lenguajes existentes diseñados para aplicaciones numéricas, con el objetivo de hacerlos aptos para programar las simulaciones discretas del mundo real".[13]

Es importante aclarar un detalle con respecto a los modernos lenguajes de programación o ambientes de programación que utilizan el término Orientado a Objetos. "Un lenguaje de

---

[13] Hanna Oktaba
Revista Soluciones Avanzadas
Número 3, Marzo-Abril 1993
Editorial Xview, S.A. de C.V.
Página 39

programación no es Orientado a Objetos sólo porque incluye la palabra reservada "objeto" o que un ambiente sea Orientado a Objetos porque tiene una interfaz agradable basada en iconos y dan la impresión de que manipulan objetos directamente". [14]

Las características más importantes que deben reunir los lenguajes de Programación Orientada a Objetos son:

1.- "Encapsulación para alcanzar la abstracción de datos.
2.- Herencia. la cual es la clave para la extensibilidad y reusabilidad.
3.- Polimorfismo y liga dinámica. Una entidad de un programa, debe permitir referenciar dinámicamente diferentes objetos y el efecto de una llamada a una rutina debe depender de la idiosincrasia del objeto que está referenciado dinámicamente".[15]

Finalmente hemos observado diferentes clasificaciones de los lenguajes de programación; por generaciones, por tipos, por utilización. Los lenguajes que nos ocupan, los Orientados a Objetos se subdividen según su forma de crear objetos en orientados a clases y orientados a objetos. Si el lenguaje permite crear objetos a partir de clases; es orientado a clases, en otro caso, si sólo crea objetos (sin utilizar clases), es orientado a objetos. Sin embargo esto no tiene mucha relevancia, ya que generalmente se les conoce a ambos como lenguajes Orientados a Objetos, sin importar cual es el medio por el cual crean los objetos. Más adelante veremos los conceptos detallados de objetos y de clase.

---

(14) Frederic Deramat
Revista Soluciones Avanzadas Número 4, Julio-Agosto 1993
Editorial Xview, S.A. de C.V. Página 38

(15) Ibid. Página 39

Hasta este punto hemos visto los distintos paradigmas de programación, y algunas de las características que debe cumplir un lenguaje de programación para considerarse 100% Orientado a Objetos, tales como Encapsulación, Herencia y Polimorfismo.

1.3.1 SIMULA

Los conceptos básicos de Orientación a Objetos fueron introducidos por primera vez en este lenguaje de programación. Todos los demás lenguajes de Programación Orientada a Objetos se derivan directa o indirectamente de SIMULA.

La historia de SIMULA comienza a finales de los 40's y principios de los 50's, cuando dos noruegos, Kristen Nygaard y Ole Johan Dahl, empleados del Departamento de Investigación de defensa Noruego, realizaron "el cálculo de la resonancia de absorción relativo a la construcción del primer reactor nuclear noruego, actividades en el campo de la simulación".[16]

A principios de los años 60's, salen al mercado dos lenguajes de programación; ALGOL 60 y SIMULA I. Pero las características estructuradas y posibilidad de recursión de ALGOL, fueron reconocidas como más fuertes y disponibles en ese momento, lo cual oculto un poco a SIMULA y su posibilidad de Orientación a Objetos. Sin embargo, a partir de 1967, SIMULA termina su período de gestación con la versión SIMULA 67, el cual se diseño como una extensión del ALGOL 60, pero con utilización de clases.

SIMULA era un lenguaje que trataba de imitar lo que sucedía en el mundo real, trabajando con entidades reales como los objetos y llegando a ser un lenguaje de propósito general ofreciendo capacidades de simulación como una aplicación de sus conceptos básicos. Las aplicaciones que se hicieron con este lenguaje se enfocaron a la simulación de procesos físicos e industriales. SIMULA es el gérmen de todos los lenguajes

---

[16] Ron Kerr
Special Silver Anniversary Suplement
SIGS Publications. Página 3

actuales más comerciales orientados a la programación con objetos, que tienen aproximádamente nueve años en el mercado del software.

Los conceptos introducidos por SIMULA son:

- Uso de clases y herencia
- Utilización de funciones virtuales (métodos)
- Establecimiento de la noción de tipo de dato dinámico y estático.
- Bibliotecas de clases en lugar de funciones.

"El ideal de un programa en SIMULA es crear modelos de algunos aspectos de la realidad"[17] SIMULA es el precursor de esta Tecnología, conceptos asociados y de todos los lenguajes de Programación Orientada a Objetos de la actualidad.

A continuación se muestran unas líneas de programa en SIMULA:

```
***INTERSECCION DE UNA LINEA CON DOS CIRCULOS***
CLASS CIRCLE(P,R); REF(FOINT)P; REAL R;
BEGIN REF(LINE)PROCEDURE INTERSECTS(C);REF(CIRCLE)C;
      BEGIN REAL R1,R2;
            IF C=/=NONE THEN
            BEGIN R1:=R**2-P.X**2-P.Y**2;
                  R2:=C.R**2-C.P.X**2-C.P.Y**2;
                  INTERSECTS:-NEW LINER(P.X-C.P.X,
                                        P.Y-C.P.Y,
                                        (R1-R2)/2);
            END;
      END***INTERSECTS***;
      IF P==NONE OR R<=0 THEN ERROR;
END***CIRCLE***
```

[17] Ibid. Página 12

1.3.2 C++

Desarrollado a principios de los 80's en los laboratorios Bell de AT&T de Bjarne Stroustrup. Se trata de una extensión del popular lenguaje C estándar, sólo se agregan conceptos de Orientación a Objetos extraídos de SIMULA, en un lenguaje existente. El operador ++ en lenguaje C incrementa una variable en uno, así C++ es el incremento de C estándar.

C++ ejemplifica lo que se conoce como Tecnología Híbrida, en la cual las características convencionales de un lenguaje de programación coexisten con las características de Orientación a Objetos.

Las características más importantes de C++ son:

- Compatibilidad con C a nivel lenguaje
- Estándar en el mercado actual
- Abstracción de Datos
- Manejo de Herencia
- Validación de datos estática
- Sobrecarga

C++ al igual que todos los demás lenguajes de Programación Orientada a Objetos se derivan de SIMULA, sin embargo, el lenguaje C estándar esta implícitamente incluido dentro del C++, lo cual continua asegurando la portabilidad de programas del C estándar a C++, sin cambios.

A continuación se muestra unas líneas de programas en C++:

```
//Construcción de un círculo en pantalla
class CIRCULO
   { private
       int pri_x,pri_y;
       int pri_radio;
       int pri_visible;
     public:
       CIRCULO(int InicX, int InicY, int InicRadio)

{if (0<=pri_x-InicRadio) && (pri_x+InicRadio<=ResX) &&
    (0<=pri_y-InicRadio) && (pri_y+InicRadio<=ResY))
    {pri_x=InicX; pri_y=InicY; pri_radio=InicRadio;
     pri_radio=0;}
//ResX y ResY son constantes con la resolucion de la
//pantalla
     }
     int X(void) { return pri_x; }
     int Y(void) { return pri_y; }
     int Radio(void) {return pri_radio; }
     int Visible { return pri_visible }
     void Muestra(void);
     void Oculta(void);
     void Traslada(int NuevoX, int NuevoY);
     };
   void CIRCULO::Muestra(void)
     { if (!pri_visible)// pintar en el color que este activo
       { pri_visible=1;
         Circle(pri_x, pri_y, pri_radio);
       };
     };
```

1.2.3 EIFFEL

Las características más importantes de Eiffel son:

- Manejo de Abstracción de Datos
- Manejo de Herencia Múltiple
- Validación estática
- Liga Dinámica

Eiffel no es muy utilizado actualmente, ya que no hay muchos compiladores comerciales de Eiffel y ninguno eficiente y accesible a las computadoras personales del gran público, sin embargo, tiene características importantes que otros lenguajes no aportan, como la Herencia Múltiple, la que detallaremos más adelante.

A continuación se muestra unas líneas de un programa en este lenguaje:

```
-- Cálculo del saldo de una cuenta
class CUENTA
            feature
              saldo_minimo:INTEGER;
              saldo:INTEGER;
              --...
              extraccion(cantidad:INTEGER) do
                require
                  cantidad>=0; saldo_minimo<=saldo-cantidad
                do
                  saldo:=saldo-cantidad
                ensure
                  saldo=old saldo-cantidad
                end;
```

1.3.4 SMALLTALK

Desarrollado a principios de los 70's, en una investigación de Xerox, en Palo Alto, California, por el equipo de investigadores dirigidos por Alan Kay. También Smalltalk se deriva de SIMULA.

El lenguaje Smalltalk desde su creación ha sufrido varias revisiones en su funcionalidad y por ello existen varias versiones, pero la mayoría de los autores consideran a Smalltalk/V como uno de los lenguajes más potentes en la programación orientada a objetos y se le atribuye este poder debido a sus características de interacción con el usuario en un ambiente puro.

El hecho de contar con rasgos puramente orientado a objetos nos refleja una forma revolucionaria de crear software, lo que significa una nueva dimensión en la que se organiza el software de forma altamente reutilizable.

Todas las versiones de este lenguaje proveen una GUI con gran variedad de textos y gráficas que pueden ser utilizados tanto para pequeñas evaluaciones como para grandes aplicaciones.

El lenguaje orientado a objetos Smalltalk es considerado como todo un ambiente, debido a que el usuario tiene un acceso directo sobre el lenguaje y manipulación sobre ciertos componentes del sistema, sin necesidad de permanecer solamente en una opción predefinida por el mismo.

Las características más importantes de Smalltalk son:

- Manejo de Abstracción de Tipos
- Manejo de herencia
- Liga dinámica
- Estándar en el mercado
- Poder para el desarrollo de sistemas de graficación
- Validación dinámica de tipos

Smalltalk tiene poco desarrollado el mecanismo de la Herencia, sin embargo, si lo puede soportar, también valida los tipos de datos al momento de su ejecución, lo que se conoce como validación dinámica de tipos.

Se presenta a continuación líneas de un programa en Smalltalk/V:

```
"Programa que abrevia un nombre largo"
|nombre long|
nombre := Prompter prompt: 'Introduzca el nombre '
            default: ''.
nombre := nombre reversed.
long   := nombre size.
^(nombre reject:[:caracter |
        caracter isVowel and:[
        c isLowerCase and: [
        (long:=long - 1) >= 8 ))))
            reversed,'
            copy from 1 to: 8.
```

1.3.5 CLIPPER 5

Clipper surgió originalmente a la sombra de DBase III, en versión compilada, tratando de superar las deficiencias que este presentaba, permitiendo además obtener directamente programas ejecutables a nivel del Sistema Operativo DOS.

Posteriormente surgieron otras versiones diferentes entre ambos. La versión mas reciente de Clipper; Clipper 5.2 se aleja radicalmente de la compatibilidad con DBase, tomando algunas características del lenguaje C, como son:

- Uso de Preprocesador
- Ficheros de cabecera
- Operadores aritméticos y lógicos de C
- Comentarios tipo C

Clipper 5 no es un lenguaje de programación 100% orientado a Objetos, es al igual que el lenguaje C++, un lenguaje híbrido, es decir, es un lenguaje altamente estructurado que combina parte de la Tecnología Orientada a Objetos. Incluye 4 clases -TBrowse, TBColumn, Get, Error- que permiten crear objetos a los que se dota de la herencia de cada uno de ellos.

Al igual que sus antecesores, la más reciente versión de Clipper, el 5.2, únicamente es utilizado en PC's, siendo el más popular manejador de Bases de Datos en este tipo de máquinas actualmente.

A continuación se proporciona una lista de algunos de los productos en general, mas populares que circulan en el mercado de la orientación a objetos:

- Borlan C++. Lenguaje orientado a objetos para trabajar bajo DOS con la librería Turbo Visión o para Windows con Object Windows.

- OpenODB de Hewlett Packard Co. Sistema manejador de bases de datos relacionales, orientado a objetos.

- Clipper versiones 5.0 en adelante de Nantucket Co. Sistema manejador de bases de datos relacionales que incluye un compilador, con orientación a objetos.

- Turbo C++ for Windows de Borland. Lenguaje orientado a objetos para trabajar bajo Microsoft Windows.

- Objectworks\C++ ParcPlace Systems. Programación orientada a objetos en C++ para trabajar con UNIX.

- Eiffel/S de SIG Computer de Alemania. Versión de Eiffel para MS/DOS.

- Object-Oriented COBOL. Lenguaje de programación orientada a objetos, de tipo híbrido.

- Oregon C++ de Taumetric. Compilador que trabaja bajo múltiples plataformas UNIX.

- Liant C++ de Liant Software. Compilador que trabaja bajo múltiples plataformas UNIX.

- Objetive-C de The Stepstone Corp. Lenguaje de programación orientada a objetos.

- Actor for Windows. Lenguaje de programación orientada a objetos que trabaja bajo Microsoft Windows.

- Microsoft C/C++ for Windows. Lenguaje de programación orientada a objetos.

- POET, siglas de Persistent Objects and Extended Database Technology de BKS software. Manejador de bases de datos que se basa en la estructura de definiciones de clases directamente de C++ y puede trabajar bajo ambientes que incluyen a MS/DOS, Windows y UNIX.

- Object Oriented Turbo Pascal por Borland. Lenguaje de programación orientada a objetos que puede correr a partir desde su versión 6.0.

- DataFlex de Data Access Corp. Sistema manejador de bases de datos orientado a objetos.

- GeODE siglas de GemStone Object Development Environment de Servio Corp. Lenguaje para el desarrollo de aplicaciones con bases de datos orientados a objetos.

1.4 LA CRISIS DEL SOFTWARE

A continuación trataremos de los problemas más importantes, a los cuales se enfrentan las organizaciones modernas en el manejo de su información, lo cual origina la crisis del Software. Analizaremos también porque la Programación Orientada a Objetos puede ayudar a combatir esta "crisis".

Las modernas organizaciones se enfrentan con un problema relativo a la cantidad de información que tienen por procesar, ya que crece más ésta, que su habilidad para procesarla. El problema no es culpa ya del Hardware, la falla está ahora en el Software. El Hardware ya es lo suficientemente rápido, para correr cualquier aplicación aceptablemente.

Las grandes organizaciones desarrollan sus sistemas de acuerdo a sus propias necesidades, sin embargo, muchas veces se trabaja con bajo presupuesto, y apresuradamente, lo que ocasiona sistemas rígidos y con errores, lo cual hace imposible hacer cambios mayores sin rediseñar totalmente los sistemas. El problema, si se analiza más a fondo, es que muy raramente se confía en lo que otro desarrollador hizo. "Todos los desarrolladores queremos asegurarnos de que algo funcione, si no está disponible el programador, diseñador o analista original, preferimos volverlo a hacer"[18] Con lo cual la reutilización de esfuerzos es prácticamente nula.

En cuanto a los mantenimientos, cada cambio funcional a un sistema ya existente implica un gran problema, ya que los sistemas están firmemente atados a los datos. Cualquier cambio en las estructuras de datos afecta a todo el sistema.

---

[18] Abel Archundia Pineda y Jorge Muñoz Márquez. Ob.Cit. Página 93.

Estos problemas, junto con las condiciones cambiantes de los negocios afectan a las organizaciones. Muchos proyectos de Software desarrollados tienen algunos de los siguientes caminos:

- Se regresan para su reconstrucción
- Se abandonan o
- Nunca son completados

El poder reutilizar una gran cantidad de código eleva la productividad en el desarrollo de nuevas aplicaciones, una vez cubierto el costo de pasar de una a otra tecnología en cuanto a aprendizaje se refiere. "El preparar a un programador en esta tecnología, y que llegue a ser productivo, puede tomar hasta 6 meses".[19]

[19] Ibid. Página 94

1.5 EL FUTURO DEL SOFTWARE

Ahora, veremos los cambios que se espera en el desarrollo del Software bajo esta metodología, así como la relación que guarda la Programación Orientada a Objetos con la interoperatividad.

El objetivo de los programadores ha sido siempre desarrollar aplicaciones de forma rápida y barata, que necesiten con el paso del tiempo, el menor esfuerzo posible en labores de mantenimiento. En la actualidad, esta emergiendo una nueva generación de productos de desarrollo de aplicaciones: las que se programan bajo la metodología de la Programación Orientada a Objetos.

Principalmente, la Programación Orientada a Objetos permite el rápido desarrollo de las aplicaciones, facilitando el mantenimiento futuro del sistema, minimizando los errores, así como agregar más módulos a la aplicación sin necesidad de rediseñar el sistema completo.

Los beneficios de la Programación Orientada a Objetos, se extenderán también en los Sistemas Operativos, los cuales se basarán en la Tecnología Orientada a Objetos, con lo cual tendrán las siguientes características:

- "Funcionarán en plataformas de hardware diferentes sin necesidad de recompilación.
- Proporcionarán el mismo nivel de servicios básicos, independientemente de la Plataforma Hardware subyacente.

- Se podrán expandir o comprimir con facilidad para satisfacer necesidades individuales, solo añadiendo o quitando módulos de Objetos".[20]

Algunos ejemplos de Sistemas Operativos Orientados a Objetos son; el Windows NT de Microsoft o el OS/400 de IBM, aunque no la utilizan al 100%, pero es un buen comienzo.

La Tecnología Orientada a Objetos ya ha cambiado la forma de construir Software, y empieza a cambiar la manera de diseñar mejor Software.

El verdadero impacto de la Programación Orientada a Objetos esta en el cambio de perspectiva del desarrollo de sistemas, que pasa de ser una colección de subrutinas a ser una industria de componentes reutilizables.

Gracias a la Programación Orientada a Objetos, en el campo de los sistemas Operativos se logrará la interoperatividad entre plataformas Hardware diferentes. Los pronósticos son contundentes, la Programación Orientada a Objetos predominará en el Desarrollo de Aplicaciones en los 90's.

---

[20] Revista PC WORD (España) Número 75, Marzo 1992
Artículo: Sistemas Operativos orientados a objetos.
Editorial IDG Comunications. Página 224

II CONCEPTOS BÁSICOS

2.1 PROGRAMACIÓN ORIENTADA A OBJETOS

Ahora trataremos brevemente los fundamentos de la Programación Orientada a Objetos, definiendo todos los elementos que conforman esta innovadora metodología de desarrollo de Software. Revisaremos los conceptos de Programación Orientada a Objetos tales como: objeto, método, mensaje, herencia, clase, polimorfismo, encapsulación y valor de retorno, entre otros.

Para comprender porque se conoce a esta metodología con el nombre de Orientada a Objetos, haremos una analogía con el mundo "real". En el mundo físico en el cual nos movemos, cualquier cosa puede considerarse como objeto, los objetos tienen características propias, como su color, su peso o su tamaño, presentan también un cierto comportamiento en respuesta a ciertos estímulos. Los objetos del mundo real se pueden utilizar una y otra vez sin que haya necesidad de volverlos a diseñar.

Los Objetos poseen un conjunto de propiedades que los definen y distinguen de los demás, algo que les da la individualidad necesaria para que el conocimiento del

hombre pueda tratar con ellos. "Los objetos son la formalización real que hace la mente del hombre".[1] En el mundo real sólo existen los objetos, sólo conocemos objetos.

Haciendo esta consideración acerca de los objetos en el mundo físico, veremos ahora una primera aproximación de la Programación Orientada a Objetos; "es una técnica de construcción de aplicaciones que funciona de forma acorde a como lo hace la mente humana en su ubicación de la realidad. El hombre conoce mediante un proceso de abstracción en el que de lo real vamos tomando sus aspectos más comunes para construir formas inteligibles por nuestro pensamiento, estos son los objetos".[2] Es decir, la Programación Orientada a Objetos se aproxima mucho a la forma de construir objetos físicos del mundo real, pero con objetos "lógicos" o "virtuales", es decir, con programas.

Lo que se pretende al aplicar la noción de objeto al campo de la informática es fundamentalmente; reducir los tiempos de desarrollo, así como aumentar la calidad y vida del Software, además de facilitar su mantenimiento.

La programación tradicional se desarrolla en base a procedimientos y datos, basta con delimitar que procedimientos actúan sobre que datos. Los datos se estructuran con el fin de que puedan ser procesados por un conjunto de procedimientos diferentes, por lo que ambas, estructuras de datos y procedimientos, están sujetos a cambios.

---

(1) Francisco Marín Quirós, Antonio Quirós Casado y Antonio Torres Lozano
Clipper 5. Referencia Rápida
Editorial Macrobit Editores, S.A. de C.V.
México, 1991. Página 112.

(2) Ibid. Página 111.

En cambio, en la Programación Orientada a Objetos, un programa es una colección de una sola entidad básica; el objeto, el cual combina los datos con los procedimientos que actúan sobre él. Durante la ejecución, los objetos reciben y envían peticiones (o mensajes) a otros objetos para ejecutar las acciones requeridas. "La organización jerárquica de los objetos en clases permite que datos y procedimientos (métodos) de una clase sean heredados por sus descendientes. Las herencias constituyen uno de los mecanismos más potentes de la Programación Orientada a Objetos".[3]

En resumen, un Objeto es una parte de una aplicación totalmente terminada y transportable, de manera que puede usarse de forma similar por otras aplicaciones. La labor de ensamblar objetos así se vuelve más reducida, al mismo tiempo que aumenta su eficiencia. Gracias a la Programación Orientada a Objetos, en lugar de "inventar la rueda" cada vez que se empieza un nuevo proyecto, simplemente se elige el conjunto de objetos que más se aproxima a las necesidades del sistema y se extiendan esos objetos para realizar el programa en cuestión.

---

[3] Francisco Javier Ceballos
Curso de Programación C++, Programación Orientada a Objetos
Editorial Addison-Wesley Iberoamericana, S.A. de C.V.
Estados Unidos, 1993. Página 309.

2.2 OBJETO

Revisaremos ahora el concepto de objeto, encapsulación y abstracción de datos. Un programa tradicional se compone de procedimientos y datos. Un programa orientado a objetos consiste solamente de objetos que contienen tanto los procedimientos como los datos. "Un objeto es una entidad que tiene unos atributos particulares, los datos, y unas formas de operar sobre ellos, los procedimientos".[4]

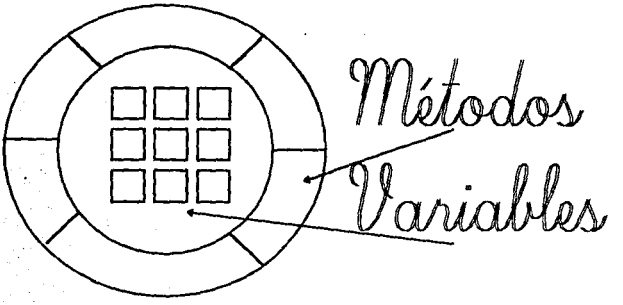


Figura 5.
Representación de un Objeto.

[4] Ibid. Página 309.

El elemento más importante de la Programación Orientada a Objetos es, precisamente el Objeto, el cual es un conjunto de código asociado con los datos operados por dicho código. Los Objetos están encapsulados, esto es, que tanto el código como los datos son inaccesibles por otros objetos.

Un Objeto se puede comparar con una célula, la cual al igual que los objetos, "combina información y comportamiento. Las células son envueltas por una membrana que solamente permite ciertos tipos de intercambios químicos con otras células".[5] Todas las interacciones entre células se dan a través de mensajes químicos reconocidos por la célula y pasados hacia el interior de ella.

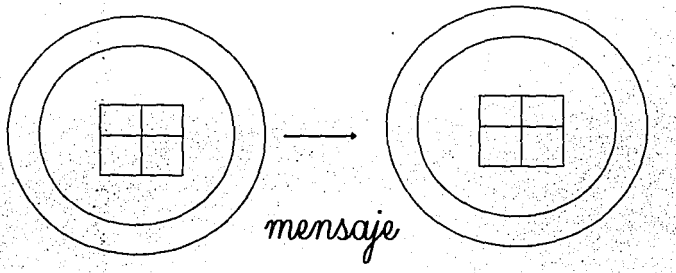


Figura 6.
Paso de mensajes entre objetos.

---

(5) David A. Taylor
Object-Oriented Tecnology: A manager guide
Editorial Adison-Wesley
Estado Unidos, 1992. Página 21

Los Objetos como los define la Tecnología Orientada a Objetos, tienen mucho de las características de las células vivientes, ya que también manejan intercambios con otros objetos, es decir, mensajes. Así pues, los mensajes permiten el intercambio de información entre objetos.

Veamos ahora una de las partes que componen al objeto, los procedimientos, técnicamente llamados métodos, que operan sobre los datos. Los datos en un objeto pueden ser accesados sólo por los métodos de ese objeto, los cuales manejan todas las tareas de rutina de los valores actuales de los datos, almacenamiento de nuevos valores o cálculos. Este arreglo entre métodos (procedimientos) y datos es llamado encapsulación. Gracias a la encapsulación se protegen los datos de la corrupción por otros objetos y oculta los detalles de implementación a bajo nivel del resto del sistema, es decir, que únicamente los métodos de un objeto pueden accesar sólo a los datos de ese mismo objeto y sólo a esos datos.

ENCAPSULACION = DATOS + PROCEDIMIENTOS

Analizaremos ahora, la otra parte de los objetos; los datos. Los datos dentro de un objeto son accesados sólo por los métodos de ese objeto. Los objetos se envían mensajes entre ellos que llaman a los métodos, los cuales, una vez activados accesan sólo a los datos requeridos. La comunicación basada en mensajes protege a los datos de un objeto de ser "corruptos" por otros objetos. Un objeto se protege así mismo de este tipo de error ocultando sus datos o, accesando estos sólo a través de sus propios métodos; es decir encapsulando datos y métodos.

La Tecnología Orientada a Objetos permite pensar al nivel del sistema del mundo real, no al nivel del lenguaje de programación que se utilizará en la aplicación. Permite definir

nuevos tipos de estructuras de datos para describir objetos del mundo real. Los modernos lenguajes de programación orientados a objetos, permiten definir nuevos tipos de datos llamados tipos de datos abstractos, por la combinación de los tipos de datos existentes en nuevas formas. Una vez definidos, nuevos tipos de datos pueden ser usados como tipos de datos básicos en un programa. Esta habilidad para crear nuevas estructuras de datos de alto nivel y usar estos en otros programas es llamada abstracción de datos.

En Programación Orientada a Objetos un objeto es una nueva estructura de datos, similar al struct de C o al record de Pascal, que contiene variables y procedimientos relacionados, denominados métodos.

Retomando el programa-ejemplo realizado en Clipper del capitulo anterior, se puede observar que para la creación de un objeto se utilizó:

```
obj:=TBrowseDB(Arriba,Izquieda,Abajo,Derecha)
columna:=TBColumnNew(ALLTRIM(TABLAS->NUMEROS)
```

Las funciones TBrowseDB y TBcolumnNew se utilizan para crear objetos.

Resumiendo, un programa tradicional se compone de procedimientos y datos, en cambio un programa orientado a objetos consiste solamente de objetos, que contienen tanto los procedimientos como los datos, sólo que encapsulados. Un objeto se entiende como una entidad que tiene atributos particulares (datos), y formas de operar sobre ellos (métodos) a través de mensajes. Los objetos son creados por medio de las clases.

2.3 MENSAJES

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a mensajes de otros objetos.

Un mensaje es el modo de comunicación con un objeto, el cual determina su comportamiento. Es decir, es la solicitud de un objeto para servirse de un segundo objeto.

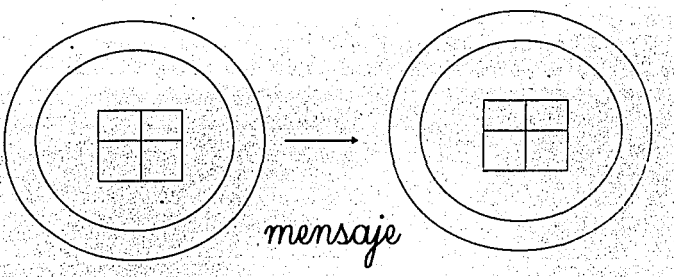


Figura 7
Comunicación entre objetos por medio de mensajes.

Los objetos se comunican entre ellos a través de mensajes. Un mensaje es simplemente el nombre de un objeto receptor, junto con el nombre de uno de sus métodos. "Un mensaje es una solicitud para el objeto receptor para llevar a cabo el método indicado y regresar el resultado de la acción".[6]

Un mensaje consiste de tres partes:

1.- Nombre del objeto receptor
2.- Nombre del método que el objeto receptor sabe como ejecutar.
3.- Parámetros que este método requiere para realizar sus funciones. Esta parte es opcional, si el método no necesita información adicional, no se necesitan parámetros en el mensaje.
Receptor:Método([Parámetros])

Sin embargo, esta estructura puede variar, ya que la manera de escribir un mensaje depende de la sintaxis del lenguaje de Programación que se esté utilizando.

Por ejemplo en el programa-ejemplo realizado en Clipper del capitulo anterior, se puede observar que para enviar mensajes se utilizó:

```
obj:HeadSep:= "=+="
obj:ColSep := "|"
```

Donde HeadSep y ColSep son los mensajes que se encuentran dentro del objeto obj y la clase TBrowse.

El conjunto de mensajes a los que un objeto puede responder se denomina protocolo. La respuesta a un mensaje es conocida como valor de retorno. Este valor de retorno puede tener diferentes formas, y depende también del lenguaje de programación que se utilice. Puede regresar un valor lógico, un

---

[6] Ibid. Página 19

número, un carácter, una cadena o nada, depende también del lenguaje de programación utilizado.

Cualquier objeto puede incluir los métodos que necesite y puede implementar estos de acuerdo a sus propias necesidades. Esto permite a cualquier mensaje ser enviado a los diferentes objetos sin preocuparse acerca de como el mensaje será manejado o conocido por la clase de objeto que los recibirá. La habilidad de ocultar los mensajes de implementación es conocida como polimorfismo. El polimorfismo hace la Tecnología Orientada a Objetos muy flexible porque permite que nuevos tipos de objetos sean agregados a un sistema completo sin reescribir los procedimientos existentes.

2.4 MÉTODO

Otra de las partes importantes que forman un objeto, son los métodos. Un método es simplemente un procedimiento o función especificado, dentro de clases de objetos, mejor conocido como subrutina o procedimiento. Este debe contener el algoritmo con los pasos necesarios que han de ejecutarse como respuesta a una mensaje y puede o no regresar un valor de retorno al mensaje que lo esta necesitando. Recordemos que los mensajes son los que activan los métodos.

Un método reside en un objeto y determina como tiene que actuar el objeto cuando recibe un mensaje. Un método puede también enviar mensajes a otros objetos solicitando una acción o información.

Por ejemplo en el programa-ejemplo realizado en Clipper del capitulo anterior, se puede observar que para ejecutar un método se utilizó:

```
obj:addColumn(columna)
```

Donde addColumn es un método que se encuentra dentro del objeto obj y de la clase TBrowse

2.5 CLASE

Una clase es una "plantilla" que se utiliza para crear objetos nuevos. Una clase en una descripción para producir objetos de esa clase o tipo. Una clase esta formada por los métodos y los datos que definen las características comunes a todos los objetos de esa clase. "La clave de la Programación Orientada a Objetos esta en abstraer los métodos y datos comunes a un conjunto de objetos y almacenarlos en una clase "[7]. Dicho de otro modo, una clase es un tipo de objeto definido por el usuario.

Analizaremos ahora un concepto más relacionado con la clase, se trata de las variables de la clase, las cuales almacenan valores que son compartidos por todos los objetos de esa clase; y cada objeto de una clase tiene sus propios valores, almacenados en las variables asociadas a cada objeto de la clase.

Entenderemos que una clase es en general un prototipo el cual describe las características de objetos similares. Los objetos pertenecientes a una clase particular son llamados instancias de clase de esa clase particular.

Las clases que se utilizaron para el programa-ejemplo del capituìo anterior fueron las clases TBrowse y TBColumn.

---

[7] Francisco Javier Ceballos. Ob.Cit. Página 307.

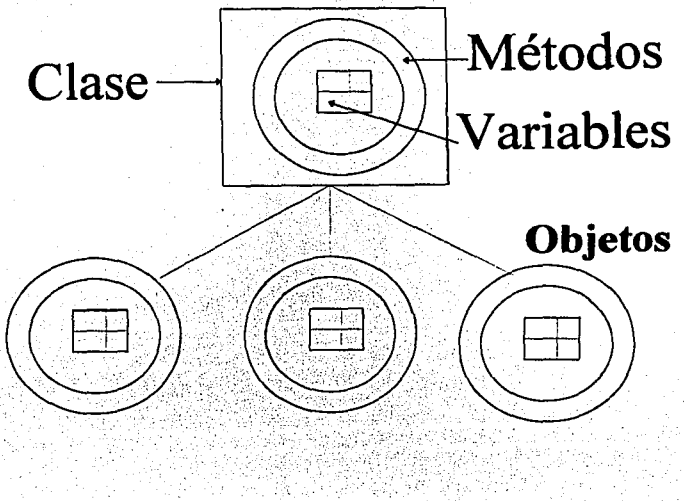


Figura 8

Esquema de una Clase.

Veamos ahora que pasa al ejecutar un objeto de una clase. Cuando un objeto recibe un mensaje para ejecutar un método que no esta definido en su clase, el objeto busca primero en su superclase, si no encuentra aquí el método, entonces busca en la superclase de su superclase y así hasta encontrarlo. Si no encuentra el método en su jerarquía de clase, entonces responde con un mensaje de error. Los objetos utilizan este mismo proceso para encontrar variables o datos.

A través del proceso llamado herencia, todas las subclases de una clase dada pueden hacer uso de los métodos y variables de esa clase. "La herencia es el mecanismo para compartir

automáticamente métodos y datos entre clases, subclases y objetos".[8]

Así pues, tanto los métodos y los datos de un objeto pueden estar distribuidos en la jerarquía de clase del objeto, esto permite responder a cualquier mensaje que solicite un método, el cual no se encuentre precisamente en el objeto, sino en su jerarquía de clase. Esta posibilidad se debe al proceso hereditario de los objetos, lo cual permite compartir métodos y datos entre clases heredadas de objetos.

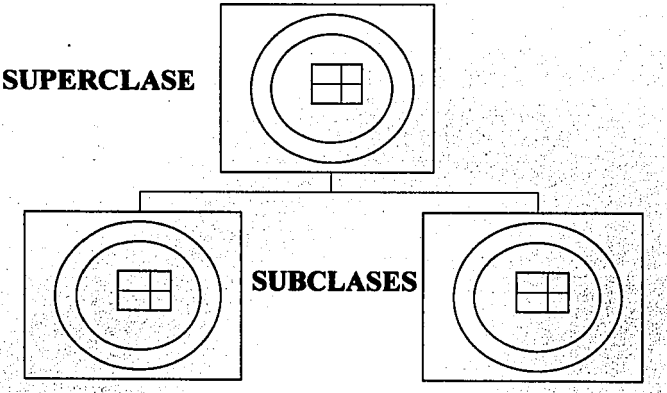


Figura 9.
Representación de la Herencia.

[8] Ibid.Página 308.

2.6 TRIÁNGULO ORIENTADO A OBJETOS

Según Bjarne Strostrup, el triángulo Orientado a Objetos, está formado por la encapsulación, tipos de datos abstractos y la orientación a objetos. Términos sobre los cuales se basa la Orientación a Objetos. De su trabajo, tenemos las siguientes definiciones:

I Encapsulación (Información Oculta)

"Decide en que módulos desea que se divida el programa, así como los datos que se ocultan en esos módulos".

II Tipos de Datos Abstractos

"Decide en que tipos desea dar un grupo lleno de operaciones para cada tipo".

III Orientado a Objeto

"Decide en que clase desea hacer común el uso de la herencia".

A continuación analizaremos detalladamente cada uno de estos tres puntos, los cuales forman el triángulo Orientado a Objetos.

2.6.1 ENCAPSULACIÓN E INFORMACIÓN OCULTA

Bjarne Stroustrup, de AT&T, desarrollador del lenguaje C++, dijo acerca de la encapsulación e información oculta, "decide en que módulos desea que se divida el programa, así como los datos que se ocultan en esos módulos". Esta es la primera diferencia del ambiente de programación estructurada, en la cual no se "ocultan" datos. En Programación Estructurada todos los datos se comparten por todos los procedimientos a través de la declaración de variables globales y del paso de parámetros por referencia.

En términos sencillos la encapsulación protege objetos contra cambios en otras partes del programa. Este mecanismo es una extensión de la estrategia de información oculta desarrollada por la Programación Estructurada. Pero en Orientación a Objetos, los datos dentro de un objeto son accesados sólo por los métodos de ese objeto.

Sin embargo, la encapsulación se da al 100% solo en los lenguajes de Programación Orientada a Objetos puros, donde no existen instrucciones o comandos como GLOBAL, PUBLIC o COMMON; clásicas de los lenguajes estructurados y por ende de los híbridos.

La encapsulación se logra con estructuras definidas en la sintaxis del lenguaje de programación, sin embargo, esta característica no existe en todos los lenguajes de programación Orientados a Objetos.

2.6.2 ABSTRACCIÓN Y CLASIFICACIÓN

Veremos ahora otros 2 importantes conceptos dentro de la programación Orientada a Objetos; la Abstracción y la Clasificación. Ambos conceptos se aplican a la Programación Estructurada, sólo que en Programación Orientada a Objetos tienen un enfoque diferente.

Bjarne Stroustrup dijo acerca de la Abstracción y Clasificación: "decide en que tipos desea dar un grupo lleno de operaciones para cada tipo".

La modularización utiliza la noción de tipo de dato abstracto, o conocido también como tipo de dato definido por el usuario. Los tipos de datos abstractos extienden la idea de los tipos de datos conocidos (entero, carácter, real, etc...) y permite al usuario definir sus propios tipos de datos.

A estos tipos de datos se les conoce como Tipos de Datos Abstractos, los cuales se pueden definir por el programador y se pueden utilizar en el programa como si fueran tipos básicos del lenguaje. "Un TDA hace a los datos lo que un procedimiento hace para las instrucciones".[9] La definición de un TDA refleja la definición de tipos de datos básicos de los lenguajes de programación: un conjunto de valores más un conjunto de operaciones que los manipulan.

---

[9] Warren R. Greiff
Revista Soluciones Avanzadas
Número 6, Noviembre-Diciembre 1993
Editorial Xview, S.A. de C.V. Página 11.

La Abstracción es por lo tanto, "un proceso cognoscitivo mediante el cual las entidades se caracterizan por propiedades de interés para un fin específico, ignorando las propiedades que no son relevantes para tal fin".[10] Así pues, el proceso de la abstracción consiste en ocultar los detalles irrelevantes con lo que además se protegen los datos, evitando que el usuario acceda a su estructura interna y la cambie sin usar las funciones especialmente diseñadas para ello, afectando así la confiabilidad del sistema.

La clasificación se construye en base a la noción de abstracción, y se "refiere a la idea de agrupar ideas de Software dentro de clases de cosas"[11]. Es decir, es necesario tratar de diferenciar cuando se habla de grupos de objetos y cuando se esta hablando acerca de objetos individuales. Esta diferenciación es importante en el Análisis y Diseño ya que puede cambiar el significado del "objeto" en alguna etapa.

---

[10] Gloria Quintanilla
Revista Soluciones Avanzadas
Número 3, Marzo-Abril 1993
Editorial Xview, S.A. de C.V. Página 44.

[11] Brian Henderson-Sellers
A book of Object-Oriented Knoledge
Editorial Prentice Hall. Serie Orientada a Objetos.
Australia, 1992. Página 23.

2.6.3 POLIMORFISMO Y HERENCIA

Revisaremos ahora otros 2 importantes conceptos de la Programación Orientada a Objetos, se trata del Polimorfismo y la Herencia. El Polimorfismo se deriva del griego y quiere decir "muchas Formas" y la Herencia es la relación que guardan las clases con respecto a otras clases. Veremos primero el Polimorfismo.

El Polimorfismo es la habilidad de referirse a los objetos de diferentes formas en tiempo de ejecución. Es decir, permite enviar cualquier mensaje a los diferentes objetos, sin preocuparse acerca de como el mensaje será manejado o conocido por la clase de objeto que lo recibirá. Es decir, todos los objetos reciben el mismo mensaje global pero pueden responder a él en formas diferentes; por ejemplo, un mensaje "+" hacia un objeto entero significa *suma*, mientras que para un objeto cadena puede significar *concatenación*.

Los beneficios que trae el polimorfismo son principalmente que hace a los objetos más independientes, además que permite agregar nuevos objetos con mínimos cambios. De esta forma se puede usar un mismo método en más de una clase de objetos.

Permite que nuevos tipos de objetos sean agregados a un sistema sin reescribir los métodos o procedimientos que existan. Así un mismo método adopta diferentes formas, en el sentido de que puede llevar a cabo diferentes actividades, dependiendo de los tipos de argumentos que recibirá.

Existen dos clases de polimorfismo: el de subtipos y el genérico. En el primero, existe la limitante de las formas que puede tomar un método, en el segundo puede adoptar un número infinito de formas.

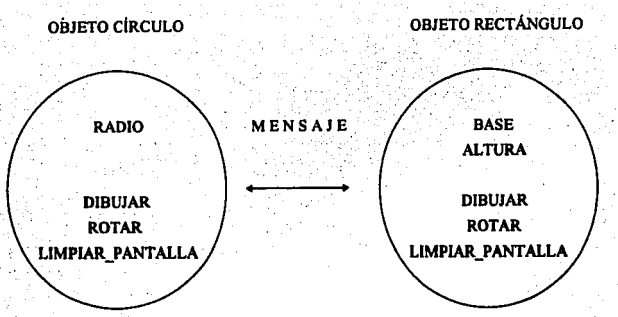


Figura 10.
Representación del Polimorfismo.

Ahora veremos el concepto de Herencia, el cual es muy parecido a la Herencia Taxonómica del mundo "real". La Herencia es una relación entre clases que permite declarar una clase (subclase) como una extensión o especialización de otra clase (superclase). A la clase que hereda se le conoce como Superclase, y a la clase heredada se le conoce como Subclase. "Los lenguajes Orientados a Objetos presentan dos variantes básicas, herencia sencilla y herencia múltiple, en la segunda, una clase puede heredar de más de una clase".[12]

---

[12] Gloria Quintanilla y Sergio Silva
Revista Soluciones Avanzadas Número 4, Julio-Agosto 1993
Artículo: Elementos de la Programación Orientada a Objetos.
Editorial Xview, S.A. de C.V. Página 5.

La herencia es el "mecanismo por el cual, una clase de objetos puede ser definida como un caso especial de una clase más general, automáticamente incluyendo las definiciones de métodos y variables de la clase general".[13] La Herencia permite a los nuevos objetos (hijos) ser derivados de unos viejos objetos (padres). Los nuevos objetos pueden tener sus propios datos u operaciones y modificar los existentes derivados de sus padres. "Las clases hijos pueden utilizar la estructura (atributos) y comportamiento (métodos) de sus clases padres".[14]

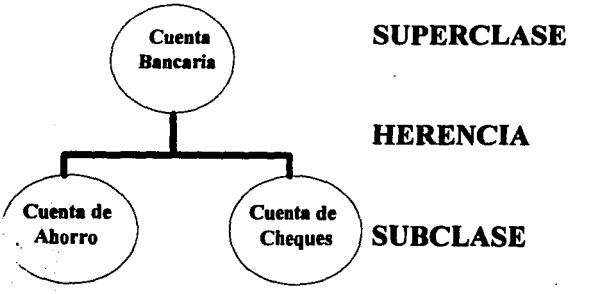


Figura 11.
Representación de la Herencia.

---

[13] David A. Taylor
Object-Oriented Tecnology: A manager guide
Editorial Adison-Wesley
Estado Unidos, 1992, Página 22.

[14] H.J. Lee & W.T. Tsai
Journal of Object-Oriented programming
Vol. 6 Number 4. Julio-Agosto de 1993
GIGS Publication. Página 53

Con la herencia se logra reducir los costos de desarrollo y eliminar las fallas incurridas al reescribir los procesos, solamente se agregan los atributos extras para las nuevas clases de objetos. La Herencia está sólo disponible en los lenguajes de Programación Orientada a Objetos, algunos de los cuales permiten que una clase herede propiedades de más de una superclase, lo cual se le conoce como Herencia Múltiple.

2.7 INTERACCIONES CON OBJETOS

Dentro de la Programación Orientada a Objetos se le conoce como interacciones con objetos a la forma de relacionar unos objetos con otros. Existen básicamente tres tipos de interacciones; la Asociación, la Agregación y la Herencia.

La Agregación representa la relación "tener a" o "consistir de", por ejemplo, una habitación consiste de cuatro paredes, un piso, etc. Es la forma natural de conceptualizar a los objetos, creados a partir de otros objetos.

La Asociación es el uso directo de los servicios de un objeto por otro, por ejemplo, un objeto cliente utiliza los servicios de un objeto banco. Esta relación se basa en la interacción entre dos objetos independientes uno de otro, los cuales se utilizan mutuamente para lograr sus objetivos.

La Herencia representa la jerarquía taxonómica o relación "es un", por ejemplo, el objeto hijo es descendiente del objeto padre. La herencia, explicada anteriormente es igual a la relación de herencia en el mundo "real", en el cual un objeto se utiliza para crear otro u otros objetos.

Sin embargo, aún no esta claramente definido en los lenguajes de programación estos conceptos, "los actuales lenguajes de Programación Orientada a Objetos no diferencian la Agregación de la Asociación".[15]

---

[15] Brian Henderson-Sellers. Ob.Cit. Página 55.

III METODOLOGÍAS

3.1 METODOLOGÍAS PARA EL DESARROLLO DE SISTEMAS DE INFORMACIÓN

Repasaremos brevemente algunas metodologías para el desarrollo de Sistemas de Información. En primer término haremos una remembranza de forma muy general del ciclo de vida "tradicional" de los sistemas.

Este ciclo se divide en tres partes principales: análisis, diseño e implementación. La forma general de dividir el ciclo de vida de los sistemas puede variar dependiendo de cada punto de vista. Algunos consideran más pasos, otros juntan varios pasos en uno o suprimen algunos pasos que consideran implícitos en otros.

A grandes rasgos, en el análisis, el problema es examinado de acuerdo a los requerimientos del usuario. En el diseño se descompone el sistema desde lo más complejo hasta lo más detallado. Posteriormente se elabora el programa de computadora, el cual es revisado y probado, si éste se encuentra aceptable se pone en uso, dándole mantenimiento regularmente.

El siguiente diagrama representa este ciclo de vida de los Sistemas:

Análisis	**Q**	**Análisis de requerimientos del usuario**
	U	**Especificaciones de requerimientos**
	E	**Especificación de requerimientos de Software.**
Diseño	**C**	**Construcción del diseño lógico**
	O	
	M	**Diseño Físico**
	O	
		Codificación
Construcción		**Pruebas a programas individuales**
		Pruebas del Sistema
		Uso del Programa
		Mantenimiento del Software

Cuadro 1.
Ciclo de Vida del Desarrollo del Sistema.
Fuente: A Book of Object-Oriented Knowledge. Página 108.

El siguiente modelo describe el ciclo de vida de los sistemas orientados a objetos, este modelo es denominado "modelo fuente", por su parecido físico a una fuente de agua.

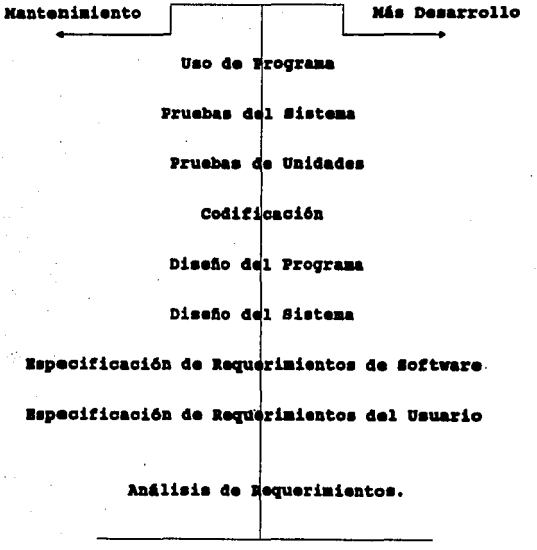


Fig. 11
Ciclo de vida del desarrollo de sistemas orientados a objetos.
Fuente: A Book of Object-Oriented Knowledge. Pág. 111

El análisis trata básicamente de estudiar el problema real y crear su modelo abstracto, incluyendo sus características más importantes. El diseño trata de darle solución a ese modelo. Si a estos procesos se les agrega las palabras Orientado a Objetos la representación y su solución deben hacerse bajo el modelo de objeto.

Por lo tanto, el Análisis Orientado a Objetos se encarga de descubrir las abstracciones que representan conceptos que tienen un significado claro, ejemplo, cliente, cajero, servicio, etc. Mientras que durante la fase del Diseño Orientado a Objetos se descubran otros objetos o abstracciones útiles para llegar a la solución.

ANÁLISIS	**Identificación de las clases de objetos semánticos, los atributos de las operaciones que describen el comportamiento de los objetos**
	Colocación de los atributos y las operaciones dentro de las clases, así como la definición de las relaciones de generalización, agregación y asociación entre clases.
	Especificación del comportamiento dinámico de los objetos.
DISEÑO	**Optimización de las clases, la cual puede consistir en la reestructuración de las jerarquías y relaciones entre clases.**

Cuadro 2.
Análisis y Diseño Orientado a Objetos.
Fuente: Revista Soluciones Avanzadas Número 4. Página 10.

El Diseño Orientado a Objetos es una técnica que se basa en la descomposición modular de un sistema en clases de objetos que el sistema manipula, en lugar de las funciones que el sistema realiza. "La descomposición funcional se basa en la identificación de las acciones a efectuar (verbos). Cada función se descompone en funciones más sencillas, hasta llegar a funciones simples. La descomposición Orientada a Objetos se efectúa identificando sustantivos, representados por objetos. Los objetos son entidades activas, capaces de efectuar acciones y de vigilar la consistencia de su estado interno"[1] Consideremos el diseño del sistema "*cocinar espagueti*", observando ambas perspectivas de diseño de sistemas.

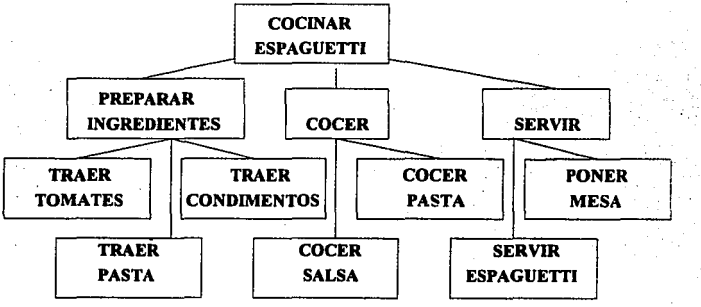


Figura 12
Diseño Funcional del Sistema "Cocinar Espaguetti".
Fuente: Revista Soluciones Avanzadas Número 3 Página 42.

---

[1] Hanna Oktaba
Revista Soluciones Avanzadas Número 3, Marzo-Abril 1993
Artículo : Programación Orientada a Objetos: moda o realidad.
Editorial Xview, S.A. de C.V.
Página 42

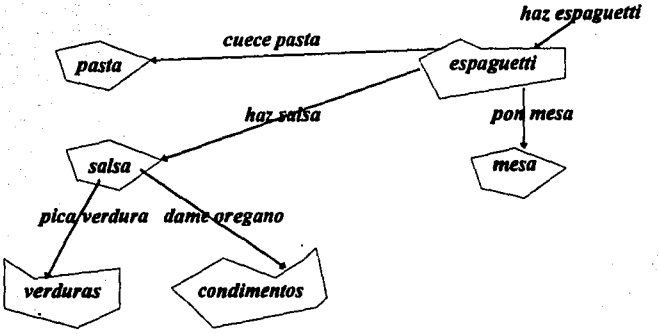


Figura 13.
Diseño Orientado a Objetos del sistema "Cocinar Espaguetti".
Fuente: Revista Soluciones Avanzadas Número 3. Página 42.

Hasta ahora hemos realizado un resumen del ciclo tradicional de desarrollo de sistemas y del ciclo de desarrollo de sistemas orientado a objetos. A grandes rasgos las metodologías se parecen, la diferencia fundamental es la aplicación del término objeto (datos con operaciones) y todos los conceptos que implica, en contraposición con la metodología tradicional o descomposición funcional en la cual las operaciones se estructuran de manera jerárquica, y los datos se encuentran en un espacio común compartido por las operaciones.

Existen algunas metodologías para el Análisis y Diseño de Sistemas Orientados a Objetos, revisaremos brevemente algunas:

METODOLOGÍA O-O-O

Análisis Orientado a Objetos
Diseño Orientado a Objetos
Programación Orientada a Objetos

- Básicamente existen dos metodologías principales de desarrollo con el enfoque (O-O-O) Orientado a objetos; la de Booch (1991) y la de Henderson-Seller y Edwards (1990).

METODOLOGÍA F-O-O

Análisis Funcional
Diseño Orientado a Objetos
Programación Orientada a Objetos

Se trata de una metodología híbrida para el desarrollo de sistemas de información, es decir, que combina metodologías de Descomposición Funcional con metodologías Orientadas a Objetos.

METODOLOGÍA O-O-F

Análisis Orientado a Objetos
Diseño Orientado a Objetos
Programación Funcional

En la metodología O-O-F, sólo se usa un modelo en el Análisis Orientado a Objetos, así como en el diseño Orientado a Objetos, además de Programación Funcional. Sin embargo, puede soportar en la Programación conceptos orientados a objetos y se puede programar en algún lenguaje de tercera generación.

METODOLOGÍAS CONVERGENTES

Esta metodología se forma por la integración de metodologías con orientación a objetos y técnicas procedurales. Permite combinar dos lenguajes de programación; uno procedural y otro orientado a objetos, aunque se diseñe el programa en términos de objetos.

Existen docenas de metologías para el desarrollo de sistemas orientados a objetos, las cuales han salido a la luz recientemente. De todas estas metodologías presentadas solamente revisaremos el detalle de la metodología O-O-O de Booch, por ser la más aceptada y utilizada.

Revisaremos a continuación los defectos de la metodología de Descomposición Funcional y un método de desarrollo de sistemas Orientado a Objetos; el Método de Booch, el cual entra en la categoría de metodologías O-O-O, es decir, tanto el Análisis, Diseño y Programación se realizan bajo el paradigma Orientado a Objetos, sin intervención de la Descomposición Funcional.

3.2 DEFECTOS DE LA METODOLOGÍA DE DESCOMPOSICIÓN FUNCIONAL

El análisis de descomposición funcional, comienza respondiendo a la interrogante ¿Qué?, es decir, ¿Qué es lo que se desea hacer?, después responde a ¿Cómo?, ¿Cómo se va a hacer?, A grandes rasgos se trata de una metodología de análisis y diseño TOP_DOWN (Arriba-Abajo).

Sin embargo, Meyer en 1954, identifica cuatro defectos en el diseño funcional TOP-DOWN:

1.- "No toma en cuenta los cambios evolutivos". Esto es, los requerimientos futuros no son contemplados en ningún momento.
2.- "Los sistemas se caracterizan por una simple función". Se hace difícil distinguir la real funcionalidad de un sistema, es decir, es difícil separarlo o tomar alguna parte de él para otra aplicación.
3.- "Se basa en el pensamiento funcional: los aspectos de la estructura de los datos son frecuentemente desatendidos". Es decir, la metodología TOP-DOWN identifica muy temprano del ciclo de vida la estructura de datos, sin siquiera haber comenzado con el análisis de la información.
4.- "No motiva el reuso". No hay motivación de reutilizar alguna parte del código creado, no es que sea imposible hacer esto, sino que no es sencillo realizarlo, ya que inicialmente no se pensó en otra aplicación más, aparte de la que se estaba desarrollando. No se piensa en la reutilización de código.

Meyer indica que inicialmente "el sistema es visualizado en un nivel alto, en términos de lo que intenta hacer y entonces,

se diseñan los detalles de como lograr el proceso orientando estas metas".[2]

Algunas herramientas de diseño que utilizan para apoyar a esta metodología son: diagramas de flujo de datos, diccionario de datos y cartas de estructura, entre otras.

La descomposición funcional es soportada y respaldada por los antiguos lenguajes orientados a procedimientos, y representa un modo natural de desarrollar sistemas en este ambiente. En oposición las metodologías de desarrollo de sistemas orientadas a objetos entre sus características principales se encuentran: promover la reutilización, considerara cambios futuros al sistema, se basa en métodos que son individuales para cada objetos, en lugar de funciones generales para todo el sistema. Los métodos para Análisis y diseño orientado a objetos proponen pasos a seguir para construir modelos en términos de clases, objetos y relaciones entre ellos.

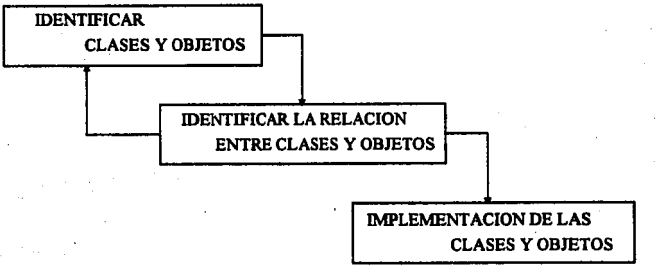
---

[2] Brian Henderson-Sellers
A book of Object-Oriented Knoledge
Editorial Prentice Hall. Serie Orientada a Objetos
Australia, 1992. Página 116.

3.3 INTRODUCCIÓN AL MÉTODO DE BOOCH

El método de Grady Booch está clasificado dentro de las metodologías O-O-O, 100% orientada a objetos, no esta mezclado con metodologías funcionales. Revisaremos sus aspectos más importantes a continuación.

Este método es descriptivo (diciendo que se podría hacer) más que prescriptivo (diciendo que se debería hacer).



PASOS DEL PROCESO

Boòch reconoce que el análisis y el diseño de un sistema no pueden realizarse aisladamente.

El proceso de análisis consiste en "descubrir las clases de objetos que modelan el dominio del problema, mientras que el diseño requiere de más invención de clases adicionales y de adaptación de lo previamente modelado".[3]

---

[3] Hanna Oktaba
Revista Soluciones Avanzadas

En el análisis se identifican los objetos y las clases que forman parte del sistema, buscando en la descripción del problema; los candidatos para las operaciones o métodos y los candidatos entre las abstracciones de clases. "Los candidatos para las clases se buscan entre los sustantivos significativos en la descripción, mientras que los candidatos para los métodos deben escogerse a partir de los verbos".[4]

Posteriormente en el diseño se definen las relaciones entre las clases basándonos en el conocimiento sobre el problema, aplicándose aquí el concepto de herencia. El descubrimiento de estas relaciones puede causar nuevos objetos y relaciones, por lo tanto estos dos pasos son iterativos hasta que se llegue a un estado satisfactorio.

Las relaciones se dividen en relaciones de uso y relaciones de herencia. "La relación de uso entre dos clases se establece cuando se descubre que una clase necesita de objetos de otra clase para realizar sus actividades. La relación de herencia entre dos clases se establece cuando una clase comparte el estado y comportamiento con otra clase más general".[5]

A continuación a cada clase establecida se le asocia una descripción más precisa, conocida como esquema (template) de clase, en el que se enlistan los métodos o servicios que ofrece cada clase, además de las subclases que contendrá.

Luego se construyen los esquemas de operaciones, que contienen la descripción de los parámetros, precondición, acción y postcondición. "La precondición define las restricciones que deben de cumplirse para que la operación se

---

Número 6, Noviembre-Diciembre 1993
Editorial Xview, S.a. de C.V. Página 18.

(4) Ibid. Página 19.

(5) Ibid. Página 19.

lleve a cabo exitosamente, la acción describe su comportamiento y la postcondición describe los resultados de la operación".[6] Esta parte es libre y puede utilizarse incluso el código del lenguaje que se usará en la implementación.

Una vez detallados todos los esquemas de operaciones, se procede a la codificación, en el lenguaje que se haya elegido.

El proceso de análisis y diseño es creativo y pueden existir algunas equivocaciones por lo que es importante realizarlo en equipo y repasarlo varias veces, ya que todo lo que aquí se resuelva será definitivo para el éxito del sistema.

Booch propone seis notaciones que permiten representar el modelo propuesto: diagrama de clase, diagrama de objetos, diagrama de sincronización, diagrama de estado, diagrama de módulos y diagrama de procesos.

DIAGRAMAS DE CLASE

Muestran las clases y las relaciones entre ellas, así como una lista de los métodos o servicios que ofrece cada clase, además de las subclases que contendrá.

Los trazos de los diagramas de clase son producidos durante los estados iniciales del desarrollo y son completados cuando se identifican la cantidad de relaciones entre las clases.

---

[6] Ibid. Página 19.

DIAGRAMAS DE OBJETOS

Muestra los objetos y las relaciones entre ellos. Mientras que las relaciones de clase son casi estáticas las relaciones de objetos son dinámicas, si durante la vida de un sistema descrito a través de pocas decenas de clases, algunos millones de objetos podrían ser creados y destruídos.

Un diagrama de objeto exhibe el comportamiento de un objeto tipico a través de mostrar los objetos y las relaciones entre ellos. Así la relación significa que los objetos pueden enviar mensajes a otros.

Los diagramas de objetos pueden ser realizados de diferentes maneras. La relacion de las lineas puede ser hecha directamente y etiquetada con los nombres de los mensajes que ellas representan. Finalmente, los mensajes pueden ser marcados con información sincronizada.

Booch hace enfásis sobre el descubrimiento los mecanismos clave de un diseño. Un mecanismo es cualquier estructura en donde los objetos trabajan juntos para dar algún comprtamiento que satisfaga un requerimiento de un problema.

Como los diagramas de clase, los trazos de los diagramas de objetos son producidos durante los estados iniciales del desarrollo y son completados cuando se identifican las relaciones entre las clases y objetos.

DIAGRAMAS DE SINCRONIZACION

Muestran el posible mensaje de comunicación entre objetos, sin mostrar el flujo de control. Los diagramas de estado muestran la información sincronizada de objetos aislados. También muestran la sincronización de los mensajes enviados, los diagramas de objeto pueden ser anotados con un información secuencial, o con un pseudocódigo que puede ser agregado a esté.

Booch sugiere utilizar los diagramas de sincronización para mostrar el orden de la información. Un diagrama de sincronización tiene al tiempo en el eje x y los diferentes objetos en el eje y, una linea representa el flujo de control entre los objetos.

La creación de un objeto y su destrucción puede ser mostrada en los diagramas de sincronización.

DIAGRAMAS DE ESTADO

Los diagramas de transición de estado muestran como las instancias de clases pueden moverse de un estado a otro bajo la influencia de eventos y muestran que acciones resultan de esos cambios de estado.

DIAGRAMAS DE MODULO Y DIAGRAMAS DE PROCESO

Booch distingue lo lógico y físico de un sistema. Los modelos descritos previamente son usados para documentar la parte lógica de un sistema, que clases existen y como colaboran entre ellas. La parte física describe los componentes del

hardware y del software de un sistema y muestra en donde las clases deberían de ser declaradas y colocadas para ser procesadas.

Los diagramas de modulo y proceso son gráficas que dan un panorama físico de un sistema en desarrollo.

Un diagrama de modulo muestra el lugar que ocupan las clases y los objetos dentro de los modulos.

Un diagrama de proceso muestra los mecanismos de los procesos (anotando que procesos estan activos), y las conexiones de comunicación entre ellos.

Los diagramas de modulo y de proceso son producidos durante la fase de implementación.

IV CLIPPER 5

4.1 OBJETOS EN CLIPPER 5

Como se menciono anteriormente Clipper 5, no es un lenguaje 100% Orientado a Objetos, es un lenguaje altamente estructurado con posibilidad de orientación a objetos.

Clipper 5 es un sistema de desarrollo de aplicaciones de bases de datos basado en un lenguaje de alto nivel. Está diseñado de tal forma que facilita su manejo por parte del usuario final o de terceros. Permite la creación y explotación de sistemas profesionales para computadoras personales compatibles y para redes locales, además que es el compilador más potente que existe actualmente en el mercado capaz de convertir los programas intérpretes de gestión de bases de datos (Dbase) en lenguaje máquina.

Se compone de un preprocesador, un compilador, un enlazador, un depurador interactivo, un editor, un generador de reportes y un manejador de bases de datos.

El Clipper apareció en el mercado en 1985, con el único próposito de servir de compilador al Dbase III plus y hasta

1990 realizó dos versiones: Autumn'86 y Summer'87 que tuvieron mucho éxito sobre todo para aquellos programadores que deseaban realizar programas profesionales.

Con la versión Summer'87 y dado que Clipper está escrito en C, se advirtió en la casa Nantucket una creciente tendencia a intentar el despegue gracias a la posibilidad que ofrecía la conexión con el lenguaje C.

Con la versión Clipper 5.2 se logra aumentar la potencia de Clipper al perfeccionar el sistema extendido para contectar con C.

El uso de sofisticadas y eficientes pantallas y la flexibilidad en el manejo de las bases de datos ha contribuido definitivamente a la difusión de Clipper entre los usuarios profesiones.

No obstante que el lenguaje de Clipper 5 no está orientado, en sí mismo, al manejo de objetos, permite la creación de objetos de alto nivel para la captura de datos, edición de bases de datos y manejo de errores.

Son cuatro las clases que incorpora Clipper 5 para trabajar con objetos; Clase Error, Clase Get, Clase TBrowse y Clase TBColumn.

Una clase es un objeto capaz de crear otros objetos de unas determinadas características relacionadas entre sí por un elevado grado de parentesco. Así por ejemplo, la clase GET creará objetos que sirvan para introducir datos, la clase ERROR se utilizará sólo para construir un programa de errores que impida la entrada de los mismos emitiendo mensajes disuasorios, la clase TBColumn permitirá la contrucción de columnas en una

tabla de datos y la clase Tbrowse creará complejas tablas capaces de mostrar un archivo de base de datos.

En Clipper 5, los objetos son conjuntos de datos con una estructura predefinida y con una serie de comportamientos ya definidos (métodos). Pero no permite la creación de nuevas clases o la declinación de nuevas clases a partir de las existentes.

CLASES ESTANDAR

Son tipos de objetos. La información que contiene un objeto y las operaciones que se le pueden aplicar dependen de la clase a la que pertence. Para cada clase existe una función de creación de objetos.

INSTANCIAS

El nuevo objeto, producto de una función de creación de una clase dada, contiene la información y se comporta de acuerdo con la clase que le haya dado origen.

Los objetos, como las matrices, se manejan por referencia; es decir, una variable del programa no puede contener un objeto, sólo puede hacer referencia a él. Cuando una variable se refiere a un objeto puede ejecutar una operación en el objeto empleando el operador de envío de mensajes ":".

VARIABLES DE INSTANCIA

Un objeto contiene toda la información necesaria para que se puedan ejecutar las operaciones previstas para su clase. Esta información se mantiene en localizaciones de almacenamiento llamadas variables de instancia.

Aquellas variables que pueden ser accedidas por el usuario se llaman variabales exportadas. Éstas pueden ser desplegadas y, en algunos casos, puede asignárseles un valor con el operador de envío de mensajes ":".

ENVIO DE MENSAJES

Las operaciones predefinidas (métodos) en un objeto, por la clase de la que proviene, se ejecutan enviando un mensaje con el operador ":". Su sintaxis es la siguiente:

```
Objeto : Mensaje ( Parámetros )
```

Para acceder a una variable de instancia exportada se emplea la sintaxis:

```
Objeto : Mensaje
```

Para asignar un nuevo valor a una variablae de instancia exportada asignable se sigue la sintaxis:

```
Objeto : Mensaje := NuevoValor
```

Los métodos se referencían como funciones.

```
método()
```

A continuación, revisaremos los mecanismos para crear clases, que incorpora el Lenguaje manejador de Bases de Datos Clipper 5.

4.1.1 CLASE ERROR

La Clase error ofrece a través de sus mensajes, información acerca de los errores producidos en tiempo de ejecución de un programa.

La forma de trabajar de esta clase es así, cuando se produce un error en la ejecución de un programa, Clipper crea un nuevo objeto ERROR que se le pasa como parámetro al bloque de código manejador de errores especificado con la función ERRORBLOCK(). Una vez en el manejador de errores, el objeto puede ser consultado para determinar el tipo de error.

FUNCION DE CLASE

ErrorNew() -> objError	Crea y regresa un nuevo objeto Error

VARIABLES DE INSTANCIA

Las variables o mensajes de la Clase Error son las siguientes:

args	Arreglo que contiene los parámetros pasados al manejador de errores.
canDefault	Valor lógico. Indica si la reconversión por defecto es posible.
canRetry	Valor lógico. Indica si es posible reintentar la operación que generó el error.

canSubstitute	Valor lógico. Indica si un resultado puede sustituirse después del error.
cargo	Variable definida por el usuario.
description	Variable de caracteres. Contiene la descripción del error.
fileName	Variable de caracteres. Contiene el nombre del archivo que produjo el error.
genCode	Variable numérica. Indica el número del error.
operation	Variable de caracteres. Descripción de la operación que produjo el error.
osCode	Variable numérica. Número de error del Sistema Operativo.
severity	Contiene un número entero que indica la severidad de la condición de error.

subCode	Variable numérica que contiene el código de error para un subsistema específico.
subSystem	Variable de caracteres que contiene la descripción del código de error proporcionado por SubCode.
tries	Variable numérica. Número de intentos de ejecución de una misma sentencia y que produjeron error.

4.1.2 CLASE GET

Clipper 5 posee un nuevo y amplio sistema para programar la edición de datos, el cual afecta directamente las sentencias GET y READ. Sin embargo, estas sentencias se pueden seguir utilizando de la misma forma que en las versiones anteriores de Clipper o de DBase, sin necesidad de utilizar esta clase.

Si se opta por programar una edición de datos propia, se puede hacer con la Clase Get, con ella se pueden crear nuevos objetos para la edición y programarlos según lo que se requiera, incluyendo la visualización, validaciones, navegación mediante teclas de movimiento del cursor, entre otras.

La edición mediante objetos se produce del siguiente modo:

1) Han de crearse tantos objetos GET como variables vayan a utilizarse.

2) El paso de la edición de un objeto, a otro se produce mediante la técnica de pasar el foco. Así, pues el GET activo, el que se esta editando, es el que posee el foco. Este se asigna mediante el método SetFocus().

FUNCION DE CLASE

GetNew([<nRenglón>],[<nCol>],[<bBloque>] -> objGet

Genera un nuevo objeto Get con las variables de insancia Get:row(<nRenglón>), Get:col, Get:picture y Get:colorSpec (<cCadenasColor>) especificadas por los argumentos proporcionados.

VARIABLES DE INSTANCIA

badDate Valor lógico. Vale .T. cuando se esta editando un valor de tipo fecha y el dato asignado es una fecha invalida. El valor .F. se produce cuando la fecha tecleada es correcta.

block Es un código de bloque que sirve para asociar un objeto GET a una variable concreta.

buffer Una variable de cadena que define el buffer de edición cuando la variable editada tiene foco.

cargo Es una variable de cualquier tipo que puede definir y emplear el usuario para enviar al objeto GET un mensaje no programado previamente mediante los métodos implícitos.

changed Es una variable lógica que vale verdadero cuando Get:buffer ha sido alterado después de recibir el foco.

clear Contiene un valor lógico que indica que la memoria intermedia (buffer) ha cambiado desde que el objeto Get realizó un proceso de entrada de datos.

col Es una variable numérica a la que se le puede asignar el número de columna de la posición donde se mostrará el GET de la pantalla.

colorSpec Es una variable de caracteres que contiene los parámetros de color necesarios para la edición del GET. El formato del contenido ha de ser el mismo que el que se asignara mediante el comando SET COLOR o a la función SETCOLOR().

decPos Variable numérica que indica la posición del punto decimal cuando la variable que se está editando es de tipo numérico.

exitState Contiene un valor número que utiliza la versión Getsys.prg para registrar la forma en que se salió del objeto Get. Las constantes manifiestas para los valores de Get:existState pueden encontrarse en el archivo Getexit.ch.

hasFocus Variable lógica que vale verdadero cuando el objeto GET al que está asociado posee el foco.

name Es una variable de caracteres que se usa para indicar el nombre de la variable que esta asociada a un objeto GET. Su uso es opcional y sólo sirve en combinación con la función READMODAL(). Cuando se crea el objeto GET no es necesario que enviemos el mensaje name.

minus Contiene un valor lógico que indica que se ha agregado un signo menos a la memoria auxilir (buffer) de edición

original	Es una variable de cualquier tipo que contiene el valor original de la variable editada en el momento en que su objeto GET asociado obtuvo el foco. Se usa para deshacer los cambios hechos en la edición de la variable.
picture	Es una variable de caracteres que contiene los parámetros PICTURE necesarios para la edición del GET. El formato del contenido ha de ser el mismo que el que se asignaría mediante la cláusula PICTURE del mandato GET.
pos	Variable numérica que contiene la posición del cursor cuando el objeto GET editado tiene el foco.
postBlock	Bloque de código usado para validar lo aditado por el objeto GET. Sirve para procesar la cláusula VALID del mandato GET. No tiene uso en la gestión del objeto GET y se emplea sólo para el sistema GET...READ estándar.
preBlock	Bloque de código usado para permitir o no la edición del objeto GET. Sirve para procesar la cláusula WHEN del mandato GET. No tiene uso en la gestión del objeto GET y se emplea sólo para el sistema GET...READ estándar.
reader	Contiene un bloque de código para instrumentar comportamientos especiales de READ para cualquier Get.

rejected Es una variable lógica que vale .T. cuando el último valor tecleado no se situó en el buffer de edición y .F. en caso contrario.

row Es una variable numérica a la que se puede asignar el número de renglón de la posición donde se mostrará el GET en pantalla.

subscript Contiene un valor numérico que define la posición del renglón en el despliegue de un objeto Get.

type Contiene una letra identificadora del tipo de dato que contiene la variable que se este editando.

typeOut Es una variable lógica que vale verdadero cuando se intenta salir del buffer de edición. Con ella podemos controlar si deseamos la salida a partir de la tecla (ENTER) o de forma automática a la última posición del buffer.

MÉTODOS

assign() Asigna a la variable editada el contenido del buffer de edición. Para que pueda tener efecto la variable ha de poseer el foco.

backspace() Borra el carácter a la izquierda del cursor, moviendo el cursor a la posición borrada.

colorDisp([<cCadenaColor>])

Cambia el color de un objeto Get y lo vuelve a desplegar. Es equivalente a la combinación de asignar Get:colorSpec y luego ejecutar Get:display.

delete() Elimina el carácter en el cursor.

delEnd() Elimina desde la posición del carácter actual hasta el final del Get.

delLeft() Borra el carácter a la izquierda del cursor.

delRight() Borra el carácter a la derecha del cursor.

delWordLeft() Borra la palabra a la izquierda del cursor.

delWordRight() Borra la palabra a la derecha del cursor.

display() Muestra en pantalla el valor del objeto GET asociado. Si dicho objeto es el poseedor del foco, se muestra también el cursor en la posición de edición que corresponda.

end() Mueve el cursor al final del buffer de edición.

home() Mueve el cursor al principio del buffer de edición.

insert(<cCarácter>)

Inserta en el buffer de edición el carácter enviado como argumento de la función insert(). El cursor se desplaza una posición a la derecha.

killFocus() Retira el foco del objeto GET que lo poseía.

left() Mueve el cursor una posición a la izquierda del buffer de edición.

overStrike(<cCarácter>)
Sitúa en el buffer de edición el carácter enviado como argumento de la función overStrike(), sobrescribiendo el que ya existía en esa posición.

reset() Reinicia los distintos parámetros asignados al objeto GET. Asimismo, vuelve a dibujar en pantalla el contenido de la variable y sitúa el cursor en la posición de inicio de la edición. Para aplicarse sobre el objeto GET, este debe poseer el foco.

right() Mueve el cursor una posición a la derecha del buffer de edición.

setFocus() Asigna al foco el objeto GET al que se asocia. Fuerza no sólo la apertura del buffer de edición, sino también la visualización del mismo en pantalla.

toDecPos() Mueve el cursor una posición a la derecha del punto decimal.

undo() Deshace los cambios realizados dejado el buffer de edición vació o con su contenido original en caso de tener algún valor implícito.

unTransform()	Convierte el valor tipo carácter en la memoria auxiliar de edición, al tipo de dato original de la variable.
updateBuffer()	Actualiza el buffer de la edición con el contenido de la variable editada. Dicha variable ha de hallarse en posición del foco.
varGet()	Devuelve el valor actual de la variable editada.
varPut()	Actualiza la variable editada con el valor enviado en varPut() como argumento.
wordLeft()	Mueve el cursor una palabra a la izquierda del buffer de edición.
wordRight()	Mueve el cursor una palabra a la derecha del buffer de edición.

4.1.3 CLASE TBROWSE

Permite realizar sofisticadas interfases de usuario para la edición de datos en forma de tablas. Para su control posee una abundante serie de mensajes y métodos. Estos objetos proveen de la infraestructura necesaria para introducir, formatear y desplegar los datos. La recuperación de datos y el posicionamiento dentro de un archivo se ejecutan gracias a un bloque de código desarrollado por el usuario. Esto último permite una gran flexibilidad y el control en el manejo de estos datos. El despliegue de datos con un cierto formato se conrola con cadenas de formato.

La utilidad del objeto TBrowse es permitir hojear el contenido de una base de datos.

FUNCIONES DE CLASE

TBrowseNew(<nArriba>,<nIzquierda>,<nAbajo>,<nDerecha>)

Genera un nuevo objeto TBrowse con las coordenadas especificadas, pero sin objetos de columna ni bloques de código para posicionar los datos de entrada.

TBrowseDB(<nArriba>,<nIzquierda>,<nAbajo>,<nDerecha>)

Genera un nuevo objeto TBrowse con las coordenadas especificadas y con bloques de código predefinidos para colocar datos en los archivos de base de datos, pero sin objetos de columna. Los bloques de código predefinidos ejecutan las operaciones GO TOP (ir al primer registro), GO BOTTOM (ir al último registro) y SKIP (cambiar de registro).

VARIABLES DE INSTANCIA

autoLite — Es una variable lógica que vale verdadero cuando el método para estabilizar la edición sobreilumina de forma automática el campo donde está posicionado el cursor.

cargo — Es una variable de cualquier tipo que puede definir y emplear el usuario para enviar el objeto Tbrowse un mensaje no programado previamente mediante los métodos implícitos.

colCount — Es una variable numérica que contiene el total de columnas que se editan.

colorSpec — Es una variable de caracteres que contiene los parámetros de color necesarios para la edición. El formato del contenido ha de ser el mismo que el que asignaríamos mediante el mandato SET COLOR o la función SETCOLOR().

colPos — Es una variable numérica que contiene el número de la columna donde está posicionado el cursor. Las columnas se numeran desde la izquierda comenzando por 1.

colSep — Contiene un valor de carácter que define el separador de la columna para las columnas TBColumnas que no contengan su propio separador.

footSep Contiene un valor tipo carácter que define el separador de pie de página para los objetos TBColumn que no tienen un separador de pie de página propio.

freeze Es una variable de caracteres que contiene el separador de columnas que se editan.

goBottomBlock Es un bloque de código que se ejecuta al llamar al método goBottom(), a fin de ejecutar una sentencia GO BOTTOM.

goTopBlock Es un bloque de código que se ejecuta al llamar al método goTop(), a fin de ejecutar una sentencia GO TOP.

headSep Es una variable de caracteres que contiene el separador para los encabezados de las columnas que se editan.

hitBottom Es una variable lógica que vale verdadero cuando se produce algún intento de salir fuera del área de datos disponible. En el proceso de estabilización, esta variable esta controlada por skipBlock.

hitTop Contiene el valor de verdad .T. si se intentó pasar del inicio de los datos disponibles, en caso contrario ontiene valor de verdad .F. Durante la fase de estabilización el objeto Tbrowse asigna esta variable si el mensaje Tbrowse:skipBlock indica que no fue posible saltar hacia atrás el número de registros especificados.

nBottom	Valor numérico asignable que identifica la fila inferior donde terminará el área de datos a editar con Tbrowse.
nLeft	Valor numérico asignado que identifica la columna izquierda donde comenzará el área de datos a editar con TBrowse.
nRight	Valor numérico asignable que identifica la columna derecha donde comenzará el área de datos a editar con TBrowse.
nTop	Valor numérico asignable que identifica el renglón superior donde comenzará el área de datos a editar con Tbrowse.
rightVisible	Contiene un valor numérico que indica la posición de la columna visible no inmovilizada más a la derecha en el despliegue de inspección (browse)
rowCount	Valor numérico que identifica el número de filas de datos visibles en la pantalla de edición. Esto no incluye cabeceras, separadores, etc.
rowPos	Valor numérico que nos informa del renglón donde se encuentra actualmente el cursor. Los renglones comienzan con el número 1 y terminan con rowCount.

skipBlock Es un bloque de código que se usa para controlar la ubicación del puntero en la edición de datos mediante TBrowse. Existe un argumento numérico que se envía al bloque de código. Este argumento representa el número de registros que serán saltados hacia adelante o hacia atrás. Cuando el argumento es cero esto significa que no habrá reposicionamiento del puntero.

stable Es una variable lógica que indica si el objeto TBrowse permanece estable o no. Este tipo de objetos permanecen estables cuando los datos están cargado y presentados en pantalla con el cursor ubicado en uno de los renglones. La estabilidad se pierde cuando se envían mensajes para variar la ubicación del cursor. Para recuperar la estabilización se usa el método stabilize().

MÉTODOS

addColumn(<objColumna>)

Añade un nuevo objeto TBColumn al objeto TBrowse. Al nombre del objeto añadido se envía como argumento de addColumn().

colorRect(<aRect>,<aColores>)

Sirve para fijar el color de un grupo de celdas entre las editadas con TBrowse. A esta función se le envían dos argumentos. Uno de ellos es un arreglo que contiene cuatro valores que indican los números de renglón y columna del comienzo y final de la zona donde se cambiará el color. El otro argumento es un arreglo con dos números correspondientes al color normal y realzado para las celdas que están dentro del grupo marcado por el otro arreglo.

colWidth(<nColumna>)

Regresa el ancho de despliegue de la columna identificada con el número <nColumna>. Si <nColumna> está fuera de rango, no se incluye o no es un número, el método regresa el valor cero.

configure() Este método sirve para reconfigurar todos los valores de los mensajes enviados al objeto TBColumn asociado.

deHilite() Elimina la sobreiluminación de la celda donde está ubicado el cursor.

delColumn(<nPosición>)

Elimina una columna del despliegue de inspección (browse). Regresa un valor que se refiere a la columna eliminada.

down() Mueve el cursor un renglón hacia abajo.

end() Mueve el cursor una columna más a la izquierda de las actualmente visibles.

getColumn(<nColumna>)
A este método se le envía un argumento numérico, devolviendo el nombre del objeto TBColumn correspondiente a la columna.

goBottom() Reposiciona los datos al principio del archivo y sitúa el cursor en el primer renglón de datos visibles, ello se produce a través de la evaluación del bloque de código goBottomblock.

goTop() Reposiciona los datos al final de archivo y sitúa el cursor en el último renglón de datos visible. Ello se produce a través de la evaluación del bloque de código goTopBlock.

hilite() Hace que se sobreilumine la celda sobre la que se halla posicionado el cursor.

home() Mueve el cursor a la columna más a la derecha de las actualmente visibles y no ancladas con freeze.

insColumn(<nPos>,<objColumna>)
Inserta un objeto columna durante el despliegue de inspección (browse). Regresa un valor de rererencia a la columna que se ha insertado.

invalidate()	Hace que la siguiente estabilización del objeto TBrowse incluyendo encabezados, vuelva a dibujar todo el TBrowse incluyendo encabezados, pies de página y todos los renglones de datos. Para forzar que se refresquen los datos, envíe un mensaje TBrowse:refreshAll().
left()	Mueve el cursor una columna a la izquierda.
pageDown()	Mueve el cursor una página abajo.
pageUp()	Mueve el cursor una página arriba.
panEnd()	Mueve el cursor a la columna de datos situada más a la derecha.
panHome()	Mueve el cursor a la columna de datos situada más a la derecha.
panLeft()	Trae columnas ocultas por la izquierda a la zona visible de la pantalla, todo ello sin reposicionar la columna donde se halla el cursor.
panRight()	Trae columnas ocultas por la derecha a la zona visible de la pantalla, todo ello sin reposicionar la columna donde se halla el cursor.
refreshAll()	Hace que vuelvan de nuevo a cargarse y mostrarse todos los datos.

refreshCurrent() Hace que vuelvan de nuevo a cargarse y mostrarse los datos del renglón sobre el que se halla el cursor.

right() Mueve el cursor una columna a la derecha.

setColumn(<nColumna>,<objNuevaCol>)
Este método sirve para cambiar el contenido de una columna entre las editadas con TBrowse. Se le envían dos argumentos. Uno sirve para indicar la posición de columna y el otro el objeto TBColumn que ubicaremos en esa posición.

stabilize() Fuerza que se produzca la estabilización. Cuando el objeto TBrowse es ya estable, este método devuelve el valor .T..

up() Mueve el cursor un renglón hacia arriba.

4.1.4 CLASE TBCOLUMN

Un objeto TBColumn es un objeto simple que contiene la información necesaria para definir completamente una columna de un objeto TBrowse. Los Objetos TBColumn no cuentan con métodos, sólo variables de instancia exportables.

Esta clase no tiene sentido sola, es una herramienta auxiliar de la clase TBrowse. Sirve para crear columnas que serán editadas mediante la clase mencionada.

FUNCION DE CLASE

TBColumnNew(<cEncabezado>,<bBloque>)

Genera un nuevo objeto TBColumn con el encabezado especificado y con un bloque de código para recuperación de datos.

VARIABLES DE INSTANCIA

block	Es un bloque de código sin argumentos que indica el nombre del campo de la base de datos que se cargara en la nueva columna creada.
cargo	Es una variable de cualquier tipo que puede definir y emplear el usuario para enviar al objeto TBColumn un mensaje no programado previamente mediante los métodos implícitos.

colorBlock	Es un bloque de código que devuelve un arreglo que contiene un par de valores numéricos de color. Puede emplearse para mostrar una columna de un color u otro dependiendo de determinadas circunstancias del contexto.
colSep	Es una variable de caracteres que contiene el separador para columnas que se editan.
defColor	Es un arreglo que contiene un par de valores numéricos de color. El primero es el del color no seleccionado (cabeceras, pies y datos no seleccionados). El segundo es el de los datos seleccionados (celda activa).
footing	Es una variable de caracteres que contiene un pie para la columna actual.
footSep	Es una variable de caracteres que contiene el separador entre las columnas y los pies fijados para las mismas.
heading	Es una variable de caracteres que contiene una cabecera para la columna actual
headSep	Es una variable de caracteres que contiene el separador entre las columnas y las cabeceras fijadas para las mismas.
width	Es un valor numérico que contiene el ancho de visualización para cada columna.

Con la incorporación de estas cuatro clases para generar objetos de la versión 5 de Clipper, se espera que en un futuro próximo se incorporen más clases y se mejore las ya existentes. Sin embargo, la incorporación de clases a un dialecto de DBase ya es un buen logro.

4.2 COMANDOS DE CLIPPER 5

A continuación revisaremos únicamente los comandos de Clipper 5 utilizados en los programas del capítulo siguiente.

```
@<nRenglón>,<nColumna>
  [SAY <exp>
      [COLOR <cCadenaColores>]
      [PICTURE <cFormatoSay>]]
  [GET <idVar>
      [COLOR <cCadenaColores>]
      [PICTURE <cFormatoGet>]]
      [WHEN <CondiciónPrevia>]
      [RANGE <fnAbajo>,<fnArriba>]
      [VALID <CondiciónPosterior>]
```

Visualiza los datos en determinada posición de la pantalla o impresora. La cláusula GET permite la digitación de una variable de memoria previamente inicializada. Puede tener el formato creado por la opción PICTURE. Con la Cláusula VALID se valida el dato que se digite.

```
@<nArriba>,<nIzquierda>
[CLEAR[TO<nAbajo>,<nDerecha>]]
```

Borra un rectángulo cuya esquina superior izquierda se especifica por el primer par de coordenadas y su esquina inferior por el segundo par de coordenadas.

```
@<nArriba>,<nIzquierda>
  TO<nAbajo>,<nDerecha>
  [DOUBLE] [COLOR <cCadenaColores>]
```

Dibuja un rectángulo sobre las coordenadas especificadas. Si se incluye la opción DOUBLE, el rectángulo será dibujado por una línea doble.

```
APPEND BLANK
```

Adiciona un registro en blanco a la base de datos. Después de la inclusión del registro en blanco, los campos vacíos deben llenarse mediante el comando REPLACE.

```
CLEAR GETS
```

Libera todos los GETS pendientes que no hayan sido recolectados por el comando READ.

```
COMMIT
```

Ejecuta la grabación de todos los buffers de archivos de las áreas actualmente seleccionadas.

```
DELETE [<rango de registros>]
   [WHILE <Condición>]
   [FOR <Condición>]
```

Marca para eliminar los registros del archivo en uso que están dentro de conjunto especificado. Estos registros no

son eliminados fisicamente hasta que se ejecuta la orden PACK y pueden ser estituidos con la orden RECALL antes de su eliminación física.

```
DO CASE
   CASE <Condición1>
      <Sentencias> ...
   [CASE <Condición2>]
   ...
      <Sentencias> ...
   [OTHER WISE]
      <Sentencias> ...
END[CASE]
```

Permite la selección de una de las posibilidades dependiendo del resultado de una prueba. Las condiciones se analizan en el orden en que se especifican.

```
[DO] WHILE <Condición>
     <Sentencias> ...
     [EXIT]
     <Sentencias>
     [LOOP]
     <Sentencias>
END[DO]
```

Ejecuta un grupo de comandos mientras se cumpla una condición como verdadera. EXIT interrumpe la ejecución de una estructura DO WHILE..ENDDO, transfiriendo la ejecución del programa a la primera instrucción después del comando ENDDO.

```
FOR <idContador>=<nInicio> TO <nTermina>
    [STEP <nIncremento>]
    <Sentencias> ...
    [EXIT]
    <Sentencias> ...
    [LOOP]
NEXT
```

Permite la ejecución de un grupo de comandos entre el comando FOR y el comando NEXT un número de veces determinado por el contador inicializado e incrementado automáticamente por el comando.

```
[STATIC] FUNCTION <idFunción>
    [(idParámetros)]
    [LOCAL<identificador>[[:=<inicializador>],...]]
    [STATIC<identificador>[[:=<inicializador>],...]]
    [FIELD<identificador>[IN<idAlias>]]
    [MEMVAR<identificadores>]
    ...
      <sentencias ejecutables>
    ...
RETURN <exp>
```

Declara una función definida por el usuario y una serie de variables locales (parámetros formales) para recibir los valores que se le pasan y las referencias. Si se declara como estática (STATIC), sólo es visible en otras funciones y procedimientos dentro del mismo archivo de programa. En caso contrario, es visible en cualquier parte del programa.

```
GO[TO]<nRegistro>|BOTTOM|TOP
```

Mueve el puntero de registros al registro indicado en el área activa o directamente al registro inicial (TOP) o al último registro (BOTTOM). Si hay algún archivo de índices abierto, el apuntador de registros se mueve de acuerdo con el índice en control de la base de datos.

```
IF <Condición1>
   <Sentencias> ...
[ELSEIF <Condición2>]
   <Sentencias> ...
[ELSE]
   <Sentencias> ...
 END[IF]
```

Evalúa una condición, y de acuerdo con el resultado de la evaluación, ejecuta una de sus dos posibles alternativas.

```
INDEX ON <expClave> TO <xIndice>
    [UNIQUE][<rango de registros>]
[FOR<Condición>]
    [WHILE<Condición>]
    [[EVAL<Condición>][EVERY<nRegistros>]]
    [ASCENDIG|DESCENDING]
```

Construye un archivo índice para un archivo en uso, cuya finalidad es ordenar la base de datos mediante el contenido de la clave suministrada. Si se especifica UNIQUE sólo se incluye en el índice el primer registro para cada valor de la clave

LOCAL <identificador> [[:=<inicializador>]...]

Declara y opcionalmente inicializa variables y arreglos locales. Una variable local puede verse solamente dentro de la función o procedimiento que la crea.

PACK

Elimina permanentemente los registros que han sido marcados para eliminación por el comando DELETE.

PRIVATE <identificador> [[:=<inicializador>]...]

Declara y, opcionalmente inicializa variables o arreglos de tipo privado, válidos en el programa donde se declaran y en los procedimientos o funciones que éste utilice.

READ [SAVE]

Permite la edición de las variables visualizados mediante el comando ..GET.

REPLACE <idCampo> WHITH <exp>
[,<idCampo2> WHITH <exp>...]
[<rango de registros>][WHILE<condición>]
[FOR<Condición>]

Sustituye el contenido de los campos especificados por los resultados o el valor de las expresiones.

RETURN [<exp>]

Retorna el control de ejecución de la línea siguiente a aquella que acciona el programa que contiene la instrucción RETURN. Si no existe un programa anterior, la ejecución retorna al DOS.

SEEK <expBúsqueda>

Busca el primer registro del archivo indexado que esté en uso o que posea la clave suministrada como argumento.

SELECT <xnAreaTrabajo>|<idAlias>

Selecciona una de las 250 áreas de trabajo de Clipper. El área de trabajo puede seleccionarse por su número o por el alias o nombre del archivo.

SET COLOR|COLOUR TO [[<estándar>]
[,<mejorado>][,<marco>][,<fondo>]
[,<noseleccionado>]]
|(<cCadenaColores>)

Determina los colores para un video policromático o los atributos para un video monocromático que Clipper podrá utilizar para la exhibición por pantalla.

SET DEFAULT TO [<xcRuta>]

Establece la unidad de disco en que todas las operaciones con archivo van a realizarse a menos que otra unidad se especifique.

SET SCOREBOARD ON|off|<xSwitch>

Determina si los mensajes de Clipper se visualizarán en la línea cero de la pantalla.

SKIP [<nRegistros>]
[ALIAS<idALIAS>|<nAreaTrabajo>]

Mueve el puntero de registro hacia adelante o hacia atrás un número especificado de registros en el área de trabajo activa o en otra.

USE[<xcBase>[INDEX<xcIndices>]
[ALIAS<xcAlias>]
[EXCLUSIVE|SHARED]
[NEW][READONLY]]
[VIA<xcControlador>]

Abre un archivo de base de datos existente, su archivo memo asociado y, opcionalmente, los archivos índices en el área de trabajo activa. Si no se especifica ningún argumento cierra el archivo de base de datos activo. EXCLUSIVE y SHARED determinan el empleo exclusivo o compartido de un archivo de base de datos en una red. NEW abre el archivo de base de datos en la siguiente área de trabajo desponible.

Si no se menciona se abre en el área de trabajo activa. READONLY establece si el archivo de base de datos sólo puede ser leído, pero no modificado. <xcControlador> es el nombre del controlador remplazable (RDD) específico para el área de trabajo activa. Si se omite se utilizará el controlador DBFNTX.

WAIT [<expMensajePetición>][TO<idVar>]

Interrumpe la ejecución de un programa hasta que alguna tecla se presione.

ZAP

Borra todos los registros de la base de datos en uso, el archivo quedará vacío.

4.3 FUNCIONES DE CLIPPER 5

&(Macrosustitución)

Sustituye el nombre de una variable de memoria por su contenido. Una macro no puede utilizarse conjuntamente con una función.

$(Comparación de cadenas)

Retorna el valor lógico .T. si la primera expresión está contenida dentro de la segunda.

```
ACHOICE(<nArriba>,<nIzquierda>,<nAbajo>,<nDerecha>,
      <acOpcionesMenú>
      [<alOpcionesSeleccionables>],
      [<cFuncionesUsuario>],
      [<nOpciónInicial>],
      [<nRenglónVentana>])
```

Permite la creación de menús rotativos (Scrolling) con la selección de los elementos mediante el movimiento del cursor o por la presión de la primera letra del argumento.

ALIAS(<nAreaTrabajo>)

Retorna el alias del archivo del área especificada. Si no especifica ningún parámetro retornará el alias del archivo del área de trabajo actual.

ALLTRIM(<cCadena>)

Elimina los blancos iniciales y finales de la expresión suministrada como argumento.

ASC(<cExpresión>)

Retorna el valor del código ASCII del primer carácter de una expresión de caracteres.

EMPTY(<exp>)

Verifica si el contenido de una variable es nulo. Si el resultado es positivo retorna .T., en caso contrario retorna .F..

EOF()

Verifica si se encontró el final del archivo, regresando .T., en caso contrario regresa .F.

EVAL(<bBloque>,[<ArgumentosBloque>])

Evalúa el código de bloque especificado, opcionalmente pasa argumentos, y arroja el valor de la última expresión dentro del bloque.

FIELDNAME(<nPosición>)

Retorna el nombre del campo correspondiente a la posición dentro de la estructura de la base de datos que esté activa.

FOUND()

Retorna el valor lógico .T. si un comando LOCATE, SEEK, CONTINUE o FIND encuentra un registro.

[I]IF(<Condición>,<expVerdadera>,<expFalsa>)

Es una prueba de condición que regresa la expresión contenida en su segundo parámetro, si la expresión lógica que se prueba es verdadera, o la expresión del tercer parámetro si es falsa.

INKEY([<nSegundos>])

Detiene el proceso hasta que se presiona una tecla o ha corrido un cierto intervalo de tiempo y retorna un número que representa la tecla presionada más recientemente. El valor retornado está entre 0 y 255, y representa el código ASCII correspondiente.

LASTKEY()

Retorna el valor ASCII de la última tecla presionada, incluyendo las teclas de control, permitiendo de este modo saber que tecla se presionó en un comando de edición.

LASTREC()|RECCOUNT()

Retorna el número de registros del archivo en uso. Permite el empleo de alias.

RECNO()

Retorna el número de registro actual del archivo en uso.

RESTSCREEN(<nArriba>,<nIzquierda>,<nAbajo>,<nDerecha>,
<cPantalla>)

Restaura una pantalla que se había grabado previamente, por medio de una función SAVESCREEN().

SAVESCREEN(<nArriba>,<nIzquierda>,<nAbajo>,<nDerecha>)

Grava una región de la pantalla en una variable de memoria. Deben suministrarse las coordenadas de la pantalla que identifican la región por grabar.

SETCOLOR([<cCadenacolor>])

Retorna una cadena que contiene la definición de los colores, activa en ese momento. Si se especifica un argumento, deben usarse sólo letras para nombrar los colores.

SETCURSOR([<nFormaCursor>])

Ajusta la forma y tamaño del cursor. <nFormaCursor> es un valor numérico entre 0 y 4. 0 desaparece el cursor. 1 hace que el cursor sea el carácter de subrayado. 2 hace que el cursor aparezca como un bloque en la mitad inferior del renglón. 3 hace que el cursor tome la forma de un bloque en toda la altura del renglón. 4 hace que el cursor aparezca como un bloque en la mitad superior de la altura del renglón.

STR(<nNúmero>,[<nLongitud>],[<nDecimales>])

Transforma el valor del primer argumento en una cadena de caracteres. La longitud de la cadena está determinada por el segundo parámetro.

SUBSTR(<cCadena>m<nInicio>,[<nNúmero>])

Extrae una parte de la expresión de caracteres, iniciando en el carácter indicado como segundo parámetro y de longitud definida como tercer parámetro.

TONE(<nFrecuencia>,<nDuración>)

Permite la programación de sonidos, se especifican la frecuencia y la duración.

UPPER(<cCadena>)

Convierte el contenido de una expresión de caracteres a letras mayúsculas.

Los comandos y funciones que aparecen aquí, no conforman la totalidad de los comandos y funciones de Clipper, sólo se incluyeron aquellos que se utilizaron en la elaboración del programa que se presenta más adelante.

La mayoría de los comandos de Clipper 5 continúan siendo estándares con respecto al DBase III+, lo que drásticamente ha cambiado son las funciones, muchas de las cuales no existen en el Dbase o algún otro de sus dialectos. Recordemos que a partir de Dbase III+ cada manejador de Bases de Datos para PC's se ha separado del estándar.

V CASO PRACTICO

PROGRAMACIÓN DE UN SISTEMA DE INFORMACIÓN, APLICANDO LA PROGRAMACIÓN ORIENTADA A OBJETOS EN LENGUAJE CLIPPER 5.2

5.1 ANÁLISIS Y DISEÑO DEL SISTEMA

El sistema a analizar se denomina "Sistema de Detección de necesidades de capacitación para el personal de una empresa", el cual es muy sencillo, básicamente se basa en la captura de los datos más relevantes de los empleados, con la finalidad de obtener información que permita diseñar planes de capacitación.

El objetivo del sistema es "Establecer un diagnóstico sobre la situación de la capacitación y detectar las necesidades en la materia del personal de la empresa, con la finalidad de implantar un sistema racional de Capacitación".

Empecemos como lo señala el método de Booch a detectar los candidatos para Clases y los Candidatos para métodos.

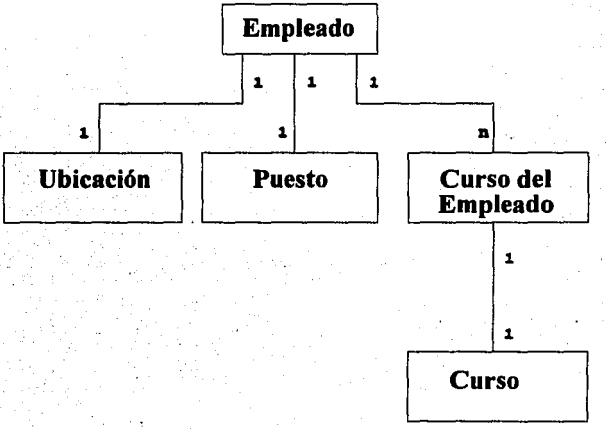
Candidatos para Clases

- Empleado
- Cursos del empleado
- Ubicación
- Puesto
- Curso

Candidatos para métodos

- Alta_Empleado
- Alta_Cursos_Empleado
- Alta_Curso
- Alta_Puesto
- Alta_Ubicación
- Borrar_Cursos_Empleado
- Borrar_Empleado
- Borrar_Puesto
- Borrar_Ubicación
- Borrar_Curso
- Buscar_Ubicación
- Buscar_Puesto
- Buscar_Curso

Considerando las clases que se han definido para el programa, lo siguiente es construir el diagrama de clases:



A continuación se deben de crear los esquemas de clase, los cuales contendrán; el nombre de la clase, un comentario acerca de lo que representa, que subclases utiliza, las operaciones o métodos que utiliza. Posteriormente vienen los esquemas de operaciones los cuales contiene el detalle de los métodos de cada clase.

Esquema de Clase

Nombre de la Clase

Empleado

Comentario

Representa la Base de datos empleado, la cual contiene los empleados de nivel operativo.

Clases que utiliza

Ubicación

Puestos

Cursos del empleado

Operaciones o métodos

Alta_Empleado

Buscar_Ubicación

Buscar_Puesto

Borrar_Empleado

Activar el objeto Cursos del Empleado

Esquema de Clase

Nombre de la Clase

Ubicación

Comentario

Representa a la Base de datos Ubicación, la cual contiene las diferentes ubicaciones o departamentos que un empleado puede tener.

Clases que utiliza

Ninguna

Operaciones o métodos

Alta_Ubicación

Borrar_Ubicación

Esquemas de Clase

Nombre de la Clase

Puesto

Comentario

Representa a la Base de datos Puestos, la cual contiene los diferentes puestos de cada empleado.

Clases que utiliza

Ninguna

Operaciones o métodos

Alta_Puesto

Borrar_Puesto

Esquema de Clase

Nombre de la Clase

Cursos del empleado

Comentario

Representa la Base de datos de todos los cursos que un empleado desea tomar. La relación con la Base de datos empleado es de un empleado a muchos cursos.

Clases que utiliza

Curso

Operaciones o métodos

Alta_Cursos_Empleado

Borrar_Cursos_Empleado

Esquema de Clase

Nombre de la Clase

Curso

Comentario

Representa la Base de datos cursos, que contiene los diferentes cursos que un empleado puede tomar.

Clases que utiliza

Ninguna

Operaciones o métodos

Alta_Curso

Borrar_Curso

Esquema de Operaciones

Nombre de la Operación

Alta_Empleado

Documentación

Permite dar de alta o modificar un empleado de nivel operativo con todos sus datos completos, siendo el método principal de todo el sistema, el cual se encargará de invocar a los otros métodos para activar otros objetos.

Precondición

Variable que indica Alta o Modificación.

Acción

```
Inicializar variables a utilizar
Repetir
     Si Alta
          Repetir
               Preguntar Clave del empleado
          Hasta que Clave no este repetida
     FinSi
     Preguntar datos generales
     Preguntar ubicación / Buscar_Ubicación
     Preguntar puesto    / Buscar_Puesto
     Alta_Cursos_Empleado
Hasta que los datos estén correctos
```

Postcondición

Ninguna

Esquema de Operaciones

Nombre de la operación

Buscar_Ubicación

Documentación

Localizar dentro de la Base de datos Ubicación la correspondiente ubicación, de acuerdo al parámetro que recibirá este método. Regresa un valor lógico verdadero si la encontró y un valor lógico falso en caso contrario.

Precondición

Clave de la ubicación.

Acción

```
Localizar la ubicación por medio del parámetro enviado
Si la encuentra
     Asignar verdadero a una variable lógica
En caso contrario
     Enviar mensaje para activar el objeto Ubicación
     Asignar falso a una variable lógica
Fin de Si
```

Postcondición

Variable lógica

Esquema de Operaciones

Nombre de la Operación

Buscar_Puesto

Documentación

Localizar dentro de la Base de datos Puestos el correspondiente puesto, de acuerdo al parámetro que recibirá este método. Regresa un valor lógico verdadero si lo encontró y un valor lógico falso en caso contrario.

Precondición

Clave del Puesto.

Acción

```
Localizar el puesto por medio del parámetro enviado
Si la encuentra
     Asignar verdadero a una variable lógica
En caso contrario
     Envía mensaje para activar el objeto Puesto
     Asignar falso a una variable lógica
Fin de Si
```

Postcondición

Variable lógica

Esquema de Operaciones

Nombre de la operación

Alta_Cursos_Empleado

Documentación

Este método se encargará de dar de alta o modificar los cursos que el empleado desee tomar, en una Base de datos temporal.

Precondición

Clave del empleado

Acción

```
Repetir
     Repetir
          Preguntar Curso
          Buscar curso
     Hasta que no esté repetido
Hasta que los datos estén correctos
```

Postcondición

Ninguna

Esquema de Operaciones

Nombre de la Operación

Buscar_Curso

Documentación

Localizar dentro de la Base de datos Cursos, el Correspondiente curso de acuerdo al parámetro que recibirá este método. Regresa un valor lógico verdadero si lo encontró y un valor lógico falso en caso contrario.

Precondición

Clave del curso.

Acción

```
Localizar el curso por medio del parámetro enviado
Si lo encuentra
     Asignar verdadero a una variable lógica
En caso contrario
     Enviar mensaje para activar el objeto Cursos
     Asignar falso a una variable lógica
Fin de Si
```

Postcondición

Variable lógica

Esquema de Operaciones

Nombre de la Operación

Borrar_Empleado

Documentación

Permite eliminar un registro de la Base de datos empleado, así como sus correspondientes cursos.

Precondición

Ninguna.

Acción

```
Pregunta si realmente se desea borrar el empleado
Si se desea borrar
     Borra el empleado indicado
     Borra los cursos asociados a ese empleado
Fin de Si
```

Postcondición

Ninguna.

Esquema de Operaciones

Nombre de la Operación

Alta_Ubicación

Documentación

Permite dar de alta una Ubicación o Departamento que no exista previamente en la Base de datos Ubicación.

Precondición

Variable que indica Alta o Modificación.

Acción

```
Inicializar variables a utilizar
Repetir
     Si Alta
          Repetir
               Preguntar Clave de la ubicación
          Hasta que Clave no esté repetida
     FinSi
     Preguntar Nombre de la ubicación o Departamento
Hasta que los datos estén correctos
```

Postcondición

Ninguna

Esquema de Operaciones

Nombre de la Operación

Borrar_Ubicación

Documentación

Permite eliminar un registro de la Base de datos Ubicación o departamentos.

Precondición

Ninguna.

Acción

```
Pregunta si realmente se desea borrar la ubicación
Si se desea borrar
     Borra la ubicación indicada
Fin de Si
```

Postcondición

Ninguna.

Esquema de Operaciones

Nombre de la Operación

Alta_Puesto

Documentación

Permite dar de alta o modificar un registro de la Base de datos Puestos.

Precondición

Variable que indica Alta o Modificación.

Acción

```
Inicializar variables a utilizar
Repetir
     Si Alta
          Repetir
               Preguntar Clave del Puesto
          Hasta que Clave no esté repetida
     FinSi
     Preguntar Nombre del Puesto
Hasta que los datos estén correctos
```

Postcondición

Ninguna

Esquema de Operaciones

Nombre de la Operación

Borrar_Puesto

Documentación

Permite eliminar un registro de la Base de datos Puesto.

Precondición

Ninguna.

Acción

```
Pregunta si realmente se desea borrar el puesto
Si se desea borrar
     Borra el puesto indicado
Fin de Si
```

Postcondición

Ninguna.

Esquema de Operaciones

Nombre de la Operación

Alta_Curso

Documentación

Permite dar de alta un Curso que no exista previamente, o modificar uno que exista de la Base de datos Cursos.

Precondición

Variable que indica Alta o Modificación.

Acción

```
Inicializar variables a utilizar
Repetir
     Si Alta
          Repetir
               Preguntar Clave del Curso
          Hasta que Clave no esté repetida
     FinSi
     Preguntar Nombre del Curso
Hasta que los datos estén correctos
```

Postcondición

Ninguna

Esquema de Operaciones

Nombre de la Operación

Borrar_Curso

Documentación

Permite eliminar un registro de la Base de datos Curso

Precondición

Ninguna.

Acción

```
Pregunta si realmente se desea borrar el Curso
SI se desea borrar
    Borra el curso indicado
Fin de SI
```

Postcondición

Ninguna.

Esquema de Operaciones

Nombre de la Operación

Borrar_Cursos_Empleado

Documentación

Permite eliminar un registro de la Base de datos Cursos del Empleado.

Precondición

Ninguna.

Acción

```
Pregunta si realmente se desea borrar el Curso
SI se desea borrar
     Borra el curso del empleado indicado
Fin de SI
```

Postcondición

Ninguna.

Ingresando a la teoría de las bases de datos, en notación relacional quedarán integradas las siguientes tablas, con sus campos correspondientes y sus llaves:

EMPLEADO (Clave empleado, Nombre, Sexo, Escolaridad.
Clave puesto, Clave Ubicación)
UBICACIÓN (Clave Ubicación, Ubicación)
PUESTO (Clave Puesto, Puesto)
CURSO (Clave curso, Nombre)
CURSO-EMPLEADO (Clave empleado, Clave curso)

A continuación veremos como quedarían estas Bases de Datos en formato XBase, definiendo el nombre que tendrá cada campo en tipo XBase, el tipo y el tamaño de cada uno de los campos de todas las bases de Datos establecidas:

EMPLEADO

Nombre XBase => *EMPLEADO.DBF*

Representación del Campo	Nombre XBase	Tipo	Tamaño
Clave del empleado	CLAVE	Numérico	5
Nombre del empleado	NOMBRE	Caracter	40
Sexo	SEXO	Caracter	1
Escolaridad	ESCOLARIDA	Caracter	1
Clave del Puesto	UBICACION	Numérico	2
Clave de la Ubicación	PUESTO	Numérico	2

UBICACIÓN

Nombre XBase => *UBICACIO.DBF*

Representación del Campo	Nombre XBase	TIPO	Tamaño
Clave de Ubicación	CLAVE_UBIC	Numérico	2
Nombre de la Ubicación	UBICACION	Carácter	30

PUESTO

Nombre XBase => *PUESTO.DBF*

Representación del Campo	Nombre XBase	TIPO	Tamaño
Clave del Puesto	CLAVE_PTO	Numérico	2
Nombre del Puesto	PUESTO	Caracter	30

CURSO

Nombre XBase => *CURSO.DBF*

Representación del Campo	Nombre XBase	TIPO	Tamaño
Clave del Curso	CLAVE_CSO	Numérico	2
Nombre del Curso	CURSO	Carácter	30

CURSO-EMPLEADO

Nombre XBase => *CSO_EMPL.DBF*

Representación del Campo	Nombre XBase	TIPO	Tamaño
Clave del Empleado	CLAVE	Numérico	5
Nombre del Curso	CLAVE_CSO	Numérico	2

Adicionalmente a las Bases de datos que utiliza el sistema para conservar la información, las cuales ya se especificaron anteriormente, tenemos tres bases de datos más. Una llamada TABLAS.DBF que conserva la definición de los campos de cada una de las Bases de datos. Los campos definidos aquí para cada Base de datos formarán parte de cada objeto creado, así como para determinar las columnas (campos de la Base de datos) que se desean desplegar en cada objeto.

La segunda Base de datos llamada ARCHIVOS.DBF contiene el nombre de cada uno de los archivos que se utilizarán en alguna aplicación en particular, así como sus índices relacionados. Al comenzar la ejecución del programa se busca en este archivo las Bases de datos que necesitará la aplicación.

Estas dos bases de Datos son indispensables para ejecutar cualquier aplicación utilizando este programa. La tercera Base de datos es opcional, pero para el caso de la aplicación que nos ocupa es requerida, se llama AUXILIAR.DBF y se utiliza para editar los cursos que cada empleado desea tomar, para después actualizar la Base de Datos de cursos del empleado.

La estructura de cada una de estas bases de datos se detalla a continuación:

TABLAS

Nombre XBase => *TABLAS.DBF*

Representación del Campo	Nombre XBase	Tipo	Tamaño
Nombre de la Base de Datos	ARCHIVO	Caracter	8
Tipo del Campo: C => Campo llave no modificable K => Campo llave modificable N => Campo no llave O => Otra función	TIPO	Caracter	1
Nombre del Campo	CAMPO	Caracter	10
Encabezado	ENCABEZADO	Caracter	25
Validación del Campo	VALID	Caracter	100
Picture	PICTURE	Caracter	20
Valor de Inicialización	INICIO	Caracter	20

ARCHIVOS

Nombre XBase => *ARCHIVOS.DBF*

Representación del campo	Nombre Xbase	Tipo	Tamaño
Nombre del archivo	ARCHIVO	Caracter	8
Tipo de Archivo: B => Base de Datos I => Indice	TIPO	Caracter	1
Llave para los archivos índice	LLAVE	Caracter	100

AUXILIAR

Nombre XBase => *AUXILIAR.DBF*

Representación del Campo	Nombre XBase	Tipo	Tamaño
Clave del Curso	CLAVE_CSO	Numérico	2

Ahora se presenta el contenido de las Bases de Datos "ARCHIVOS" y "TABLAS", para la realización de esta aplicación en particular. Anteriormente se definió cuales serían las Bases de Datos que requiere esta aplicación.

"ARCHIVOS"

ARCHIVO	TIPO	LLAVE
EMPLEADO	B	
EMPLEADO	I	CLAVE
PUESTO.	B	
PUESTO	I	CLAVE_PTO
UBICACION	B	
UBICACION	I	CLAVE_UBIC
CSO_EMPL	B	
CSO_EMPL	I	STR(CLAVE,5)+STR(CLAVE_CSO,2)
AUXILIAR	B	
AUXILIAR	I	CLAVE_CSO
CURSO	B	
CURSO	I	CLAVE_CSO

TABLAS

ARCHIVO CAMPO	T I P O	ENCABEZADO VALID PICTURE INICIO
EMPLEADO CLAVE	C	"CLAVE DEL TRABAJADOR" Xclave > 0 ("99999") 0
EMPLEADO NOMBRE	N	"NOMBRE" .T. REPLICATE ("!",40) SPACE (40)
EMPLEADO SEXO	N	"SEXO" Chk_satos(Xsexo, {"Masculino","Femenino"}) ("!") " "
EMPLEADO ESCOLARIDA	N	"ESCOLARIDAD" Chk_Datos (Xescolarida, {"A.Primaria", "B.Secundaria", "C.Comercial", "D.Bachillerato", "E.Licenciatura", "F.Posgrado", "G.Maestría", "H.Doctorado"}) ("!") " "
EMPLEADO UBICACION	N	"UBICACION" Buscar ("UBICACION", XUbicacion) ("99") 0
EMPLEADO PUESTO	N	"PUESTO" Buscar ("PUESTO", XPuesto) ("99") 0
EMPLEADO CLAVE_PTO		"CLAVE DEL PUESTO" Xclave_pto > 0 ("99") 0

ARCHIVO CAMPO	T Y P O	ENCABEZADO VALID PICTURE INICIO
PUESTO PUESTO	N	"PUESTO" .T. REPLICATE ("!",30) SPACE (30)
UBICACIO CLAVE_UBIC	C	"CLAVE DE UBICACION" XClave_Ubic > 0 ("99") 0
UBICACIO UBICACION	N	"UBICACION" .T. REPLICATE ("!",30) SPACE (30)
CURSO CLAVE_CSO	C	"CLAVE DEL CURSO" XClave_Cso >0 ("99") 0
CURSO CURSO	N	"CURSO" .T. REPLICATE ("!",30) SPACE (30)
CSO_EMPL CLAVE	C	"CLAVE DEL TRABAJADOR" Buscar ("EMPLEADO", XClave) ("99999") 0
CSO_EMPL CLAVE_CSO	N	"CLAVE DEL CURSO" Buscar ("CURSO", XClave_Cso) ("99") 0
AUXILIAR CLAVE_CSO	K	"CLAVE DEL CURSO" XClave_Cso > 0 ("99") 0

5.2 PROGRAMACIÓN DEL SISTEMA DE INFORMACIÓN, APLICANDO LA PROGRAMACIÓN ORIENTADA A OBJETOS.

De acuerdo al diseño previamente presentado, a continuación se realiza una breve descripción de los programas que aparecen posteriormente. El programa principal llamado MAIN.PRG se encarga de configurar el ambiente con el que se trabajará, además de definir todas las Bases de Datos con los índices asociados que necesitará la aplicación, de acuerdo a los datos especificados en la base de datos "ARCHIVOS". Posteriormente invoca a la función Hojear() con el parámetro "EMPLEADO", que es el primer objeto que se creará.

A continuación esta función se encarga generar el objeto del primer parámetro que se le envíe, en este caso es "EMPLEADO", es decir, el objeto empleado. Cuando se crea el objeto quedan disponibles algunas teclas para manejarlo, tales como ENTER, INSERT, DELETE, BACK SPACE, ESCAPE y teclas de movimiento del cursor. La definición de las teclas depende del "modo" con el se quiera operar el Browse o la tabla, el cual se especifica como el último parámetro que recibe la función Hojear(). Para el caso del objeto empleado, este queda definido para aceptar todas las teclas disponibles.

Las teclas importantes para la operación del objeto Browse son: ENTER, INSERT, BACK SPACE y DELETE, las cuales se encargan del mantenimiento del objeto Browse activo. Al presionar INSERT, se activa la función o método ALTA, la cual permite agregar o modificar un registro de la Base de datos activa. En este caso será para dar de alta a un empleado de nivel operativo. Siendo este el método principal de todo el sistema, se encargará de invocar a los mensajes que activan los demás objetos que se utilizan. Para el caso del objeto empleado, la

activación de los demás objetos se realiza al preguntar por la ubicación, si ésta no se encontrase, se activa inmediatamente el objeto correspondiente para poder checar los registros de ese objeto, y en su defecto agregar la ubicación no encontrada. Lo mismo pasa con los puestos y los cursos. La tecla BACK SPACE se utiliza de forma similar, la diferencia es que esta tecla, se utiliza para realizar las modificaciones al registro donde se encuentre el cursor del objeto Browse activo. Al realizar una modificación no se podrán cambiar los campos clave, solamente los campos generales.

La tecla ENTER sirve para consulta de los datos generales de cualquier objeto, presentará los datos del registro donde se encuentre el cursor del objeto Browse activo. Con la tecla DELETE se borra el registro donde se encuentre el cursor en ese momento. Con la tecla ESCAPE se termina la consulta de cualquier objeto que esté en ese momento activo, regresando al nivel anterior de donde se creo.

Las teclas de movimiento del cursor, y otras teclas en combinación con estas se utilizan para mover el cursor por todo el objeto browse activo. Para mayor explicación de las teclas de movimiento del cursor y sus posibles combinaciones, al ejecutar el programa presione la tecla F1, con la cual aparecerá la ayuda general acerca de las teclas disponibles del objeto Browse.

La función Hojear() se utiliza de forma recursiva permitiendo tener varios objetos creados en la pantalla, dependiendo del primer parámetro que se envíe para crear ese objeto, el cual debe ser el nombre de una Base de Datos, la cual debe estar previamente abierta.

Además utiliza el Polimorfismo ya que puede comportarse de diferente forma de acuerdo a los parámetros enviados.

El programa MANTO.PRG contiene las funciones o métodos necesarios para realizar el mantenimiento de cualquier Base de Datos. Incluye las funciones Alta() y Consulta().

En el programa GRALES.PRG, se incluyen las funciones o métodos generales para crear nuevos objetos si es necesario, para localizar claves dentro de listas de valores, para desplegar mensajes por pantalla y para desplegar la ayuda de cualquier objeto.

El programa OTRAS.PRG contiene funciones o métodos no estándar para todos los objetos. Se programó exclusivamente para esta aplicación, por lo cual no es aplicable a otra. Exceptuando la función Borrar, la cual debe ser modificada para que pueda ser utilizada por otra aplicación, ya que se utiliza al presionar la tecla DELETE, desde cualquier objeto activo.

Este programa se puede utilizar para cualquier aplicación sencilla. Solamente basta con definir en la Base de Datos "ARCHIVOS". Los archivos de Bases de datos e índices que se utilizarán, y en la Base de datos "TABLAS", todos los campos de esas Bases de datos y la relación que guardan entre sí estos campos con campos de otras Bases de Datos. Si se utiliza alguna base de datos que guarde sólo relaciones como en el caso de

"CSO_EMPL" se deberá programar las funciones para conservar estas relaciones.

A continuación se muestran los listados de estos programas fuente en Clipper 5.2

```
//
// Proyecto  : Sistema ejemplo para Tesis
// Programa  : MAIN.PRG
// Funciones : Main()
//
// FUNCTION Main (Path1, Path2, Dbf )
// Recibe :     1.- Directorio donde se encuentran los archivos
//                  de control ("ARCHIVOS" Y "TABLAS").
//              2.- Directorio donde se encuentran los archivos
//                  de la Aplicación.
//              3.- Nombre del primer objeto que se creará.
// Regresa : Nada
// Se encarga de configurar el ambiente con el que se trabajará,
//  además de definir todas las Bases de Datos con los índices
//  asociados que necesitan. Llama a la función Hojear() con
//  el tercer par metro que reciba, el cual será el primer
//  objeto que se creará.
//
local area, alias
path1 = "c:\silvia"
path2 = "c:\silvia"
dbf = "EMPLEADO"

CLEAR

 SET DEFAULT TO &Path1
 SET SCOREBOARD OFF

 SELECT 1
 if file("archivos.ntx")
   USE ARCHIVOS INDEX ARCHIVOS
 else
   use archivos
   index on archivo to archivos
 endif
 GO TOP
```

```
// Abre las Bases de Datos de la Aplicación
store 1 to cc

DO WHILE .NOT. EOF()
 Area  := ARCHIVOS->ARCHIVO
 Alias := IF(.NOT. EMPTY( ARCHIVOS->LLAVE ), ARCHIVOS->LLAVE, Area)
 SET DEFAULT TO &Path2
 USE &Area NEW ALIAS &Alias
 SELECT 1
 SKIP
 // Abre los Indices de las Bases de Datos
 DO WHILE ARCHIVOS->TIPO = "I" .AND. .NOT. EOF()
  SELECT &Alias
  Area := ARCHIVOS->ARCHIVO
  SET INDEX TO &Area
  SELECT 1
  SKIP
 ENDDO
ENDDO


SELECT 1
SET DEFAULT TO &Path1
if file("tablas.ntx")
 USE TABLAS INDEX TABLAS
else
 use tablas
 index on archivo to tablas
endif

SETCOLOR("N/BG")
CLEAR SCREEN

Hojear( Dbf, 5, 4, MaxRow()-5, MaxCol()-4, 1 )

CLEAR SCREEN

RETURN NIL
```

```
// Proyecto  : Sistema ejemplo para Tesis
// Programa  : CLASES.PRG
// Funciones : Hojear()
//
FUNCTION Hojear( Fdbf, Arriba, Izquierda, Abajo, Derecha, Modo )
// Recibe :     1.- Base de datos sobre la cual creará el objeto
//              2.- Renglón superior para desplegar el objeto
//              3.- Columna superior para desplegar el objeto
//              4.- Renglón inferior para desplegar el objeto
//              5.- Columna inferior para desplegar el objeto
//              6.- Modo del Browse :    1 => Todas las teclas
//                                       2 => Excepto Enter
//                                       3 => Ninguna
// Nota: Las teclas referidas anteriormente son Enter,
//   BackSpace, Insert & Delete.
//   Independientemente del modo todos los Browses tienen
//     las teclas de movimiento del cursor activas.
// Regresa : Nada
// Se encarga generar el objeto del primer parámetro que se le
//   envie, sobre las coordenadas de los siguientes parámetros.
//   La Base debe de estar abierta previamente.
//   Al momento de desplegarse el objeto las teclas quedan
//   definidas de la siguiente manera:
//   Sólo en Modo 1 y 2
//     Tecla INSERT                    => Dar de Alta registros
//     Tecla DELETE                    => Borrar el registro actual
//     Tecla BACK SPACE                => Modificar registro actual
//   En todos los Modos
//     Tecla FLECHA ABAJO              => Mover el cursor hacia abajo
//     Tecla FLECHA ARRIBA             => Mover el cursor hacia arriba
//     Tecla PAGINA ARRIBA             => Mover una página hacia arriba
//     Tecla PAGINA ABAJO              => Mover una página hacia abajo
//     Tecla CTRL_REPAG                => Mover cursor al primer registro
//     Tecla CTRL_AVPAG                => Mover cursor al último registro
//     Tecla FLECHA DERECHA            => Mover cursor hacia la derecha
//     Tecla FLECHA IZQUIERDA          => Mover cursor hacia la izquierda
//     Tecla INICIO                    => Mover el cursor a primera
//                                        columna visible
//     Tecla FIN                       => Mover el cursor a última
//                                        columna visible
```

```
//    Tecla CTRL_INICIO                    => Mover el cursor a la primera
//                                            columna
//    Tecla CTRL_FIN                       => Mover el cursor a la última
//                                            columna
//    Tecla ESCAPE                         => Finalizar despliegue del objeto
//    Tecla F1                             => Ayuda
//    Tecla INTRO                          => Para el Modo 1 consulta
//                                       Para el Modo 2 y 3 regresa a lo anterior
//

LOCAL b, Columna, n, Curcolor, CurCursor, Mas, Tecla, Y
PRIVATE X := 0
PRIVATE Arreglo := ARRAY(15, 6)

 b := TBrowseDB( Arriba, Izquierda, Abajo, Derecha )
 b:HeadSep := "=+="
 b:colSep  := " | "
 b:colorSpec := "N/W, BG/B, B/W, B/BG, B/W, N/BG, R/W, B/R"
 SELECT 1
 SEEK Fdbf

 // Carga las columnas del Browse
 WHILE Fdbf == ALLTRIM( TABLAS->ARCHIVO )
  Columna := TBColumnNew( ALLTRIM( TABLAS->ENCABEZADO ), ;
        FieldWBlock( TABLAS->CAMPO, SELECT(Fdbf) ) )
  Y := 1
  Arreglo[ ++X, Y++ ] := TABLAS->CAMPO
  Arreglo[ X, Y++ ]   := TABLAS->ENCABEZADO
  Arreglo[ X, Y++ ]   := TABLAS->TIPO
  Arreglo[ X, Y++ ]   := ALLTRIM( TABLAS->VALID )
  Arreglo[ X, Y++ ]   := TABLAS->PICTURE
  Arreglo[ X, Y ]     := TABLAS->INICIO

  IF TABLAS->TIPO <> "O"
   b:addColumn(Columna)
  ENDIF

  SELECT TABLAS
  SKIP
 ENDDO
```

```
SELECT &Fdbf

b:freeze := 1

// Dibuja sombra
CurColor := SETCOLOR("N/N")
@Arriba+1, Izquierda+1 CLEAR TO Abajo+1, Derecha+1
SETCOLOR("W/W")
@Arriba, Izquierda CLEAR TO Abajo, Derecha
SETCOLOR(CurColor)
CurCursor := SETCURSOR(0)

Mas := .T.
WHILE (Mas)

 IF b:colPos <= b:freeze
   b:colPos := b:freeze+1
 ENDIF

 WHILE !b:stabilize()
  Tecla := INKEY()

  IF Tecla == 0
    EXIT
  ENDIF

 ENDDO

 IF b:stable
  IF b:hitTOP .OR. b:hitBottom
    TONE(125, 0)
  ENDIF
  Tecla := INKEY(0)
 ENDIF

 DO CASE
  CASE Tecla == 22 .AND. Modo <> 3 // INSERT
    Alta( Fdbf, .F. )
    b:Refreshall()
```

```
CASE Tecla == 7 .AND. Modo <> 3 // DELETE
 IF LASTREC() <> 0
  Borrar (Fdbf)
  b:Refreshall()
 ENDIF
CASE Tecla == 8 .AND. Modo <> 3 // BACK SPACE
 IF LASTREC() <> 0
  Alta( Fdbf, .T. )
  b:Refreshall()
 ENDIF
CASE Tecla == 24 // FLECHAB
 b:down()
CASE Tecla == 5 // FLECHAR
 b:up()
CASE Tecla == 3 // AVPAG
 b:pageDown()
CASE Tecla == 18 // REPAG
 b:pageUp()
CASE Tecla == 31 // CTRL_REPAG
 b:goTop()
CASE Tecla == 4 // FLECHAD
 b:right()
CASE Tecla == 19 // FLECHAI
 b:left()
CASE Tecla == 1 // INICIO
 b:home()
CASE Tecla == 6 // FIN
 b:end()
CASE Tecla == 29 // CTRL_INICIO
 b:panHome()
CASE Tecla == 23 // CRTL_FIN
 b:panEnd()
CASE Tecla == 27 // ESCAPE
 Mas := .F.
CASE Tecla == 28 // F1
 Ayuda( Modo )
CASE Tecla == 13 // INTRO
 IF Modo = 1
  IF LASTREC() <> 0
   Consulta( Fdbf )
  ENDIF
 ELSE
  Mas := .F.
```

```
    ENDIF
  ENDCASE
 ENDDO
 SETCURSOR(CurCursor)

RETURN NIL
#command REPEAT                    => WHILE .T.
#command UNTIL <exp>               => IF <exp> ; EXIT ; END ; END
```

```
//
// Proyecto  : Sistema ejemplo para Tesis
// Programa  : OTRAS.PRG
// Funciones : Cursos()
//             Carga_Cursos()
//             Salvar_Cursos()
//             Borrar()
//

FUNCTION Cursos( Empleado )
// Recibe  : 1.- Clave del Empleado
// Regresa : Nada
// Función no estándar en el programa, se realizó para poder
// editar
// los cursos de cada Empleado. Se ejecuta cuando se llegue
// al campo tipo "O" del archivo "EMPLEADO".

LOCAL XAlias := SELECT()
LOCAL Modo   := IF( LASTKEY() = 13, 3, 2 )

 Carga_Cursos( Empleado )

 REPEAT
   Hojear( "AUXILIAR", 15,50, MaxRow()-3, MaxCol()-10, Modo )
 UNTIL  Modo = 3 .OR. LASTKEY() = 277 .OR.;
      MENSAJE("DATOS CORRECTOS (S/N)", "SN" ) = "S"

 IF Modo = 2
   Salvar_Cursos( Empleado )
 ENDIF

 SELECT ( XAlias )

RETURN NIL

FUNCTION Carga_Cursos( Empleado )
// Recibe  : 1.- Clave del Empleado
// Regresa : Nada
// Carga a una base de datos auxiliar los cursos correspondientes
//  al Empleado enviado como primer parámetro
```

```
SELECT AUXILIAR
ZAP
SELECT CSO_EMPL
SEEK " " + STR(Empleado, 5)

DO WHILE CSO_EMPL->DELETED <> "X" .AND. ;
      Empleado = CSO_EMPL->CLAVE .AND. !EOF()
 SELECT AUXILIAR
 APPEND BLANK
 REPLACE AUXILIAR->CLAVE_CSO WITH  CSO_EMPL->CLAVE_CSO
 SELECT CSO_EMPL
 SKIP
ENDDO

SELECT AUXILIAR
GO TOP

RETURN NIL

FUNCTION Salvar_Cursos( XClave )
// Recibe  : 1.- Clave del Empleado
// Regresa : Nada
// Guarda a una base de datos auxiliar a la base de datos de
//  cursos del Empleado, los cursos asignados a éste.

SELECT AUXILIAR
GO TOP
SELECT CSO_EMPL
SEEK " " + STR(XClave,5)
SELECT AUXILIAR

DO WHILE CSO_EMPL->DELETED = " " .AND. ;
      XClave = CSO_EMPL->CLAVE .AND. !EOF()
 SELECT CSO_EMPL
 REPLACE CSO_EMPL->CLAVE    WITH  XClave
 REPLACE CSO_EMPL->CLAVE_CSO WITH  AUXILIAR->CLAVE_CSO
 SKIP -1
```

```
 IF CSO_EMPL->CLAVE = XClave .AND. ;
   CSO_EMPL->CLAVE_CSO = AUXILIAR->CLAVE_CSO
   SKIP
 ELSE
   SKIP 2
 ENDIF
 SELECT AUXILIAR
 SKIP
ENDDO

SELECT AUXILIAR

IF !EOF()

 DO WHILE !EOF()

   SELECT CSO_EMPL
   SEEK "X"

   IF !FOUND()
    APPEND BLANK
   ELSE
    REPLACE CSO_EMPL->DELETED  WITH " "
   ENDIF
    REPLACE CSO_EMPL->CLAVE    WITH  XClave
    REPLACE CSO_EMPL->CLAVE_CSO WITH  AUXILIAR->CLAVE_CSO
    SELECT AUXILIAR
    SKIP

  ENDDO

 ELSE

   SELECT CSO_EMPL
```

```
   DO WHILE XClave = CSO_EMPL->CLAVE .AND. !EOF()
    SKIP
    Pointer := RECNO()
    SKIP -1
    REPLACE CSO_EMPL_->DELETED  WITH "X"
    GO Pointer
   ENDDO

  ENDIF

  SELECT AUXILIAR
  ZAP

RETURN NIL

FUNCTION Borrar( Fdbf )
// Recibe  : 1.- Base de datos sobre la cual borrar  un registro
// Regresa : Nada
// Permite borrar un registro de la base de datos especificada,
//  Si se está operando sobre la base de datos Empleado
//  borrará a éste incluyendo sus cursos.

LOCAL Pointer :=0

 IF MENSAJE( "DESEA BORRAR ESTE REGISTRO (S/N)", "SN" ) = "S"

  // Condición aplicable únicamente para esta aplicación
  IF Fdbf == "EMPLEADO"
   SELECT CSO_EMPL
   SEEK " " + STR(EMPLEADO->CLAVE, 5)

   DO WHILE EMPLEADO->CLAVE = CSO_EMPL->CLAVE .AND. !EOF()
    SKIP
    Pointer := RECNO()
    SKIP -1
    REPLACE CSO_EMPL->DELETED  WITH "X"
    GO Pointer
   ENDDO

  ENDIF
```

```
   SELECT &Fdbf
   DELETE
   PACK
   CCMMIT

  ENDIF

RETURN NIL
#command REPEAT                 => WHILE .T.
#command UNTIL <exp>            =>IF <exp> ; EXIT ; END ; END
```

```
//
// Proyecto  : Sistema ejemplo para tesis
// Programa  : MANTO.PRG
// Funciones :  Alta()
//              Consulta()
//

FUNCTION Alta( Fdbf, Mod )
// Recibe :          1.- Alias de la Base de Datos activa
//                   2.- Tipo de Alta :    Falso        => Nueva Alta
//                                         Verdadero    => Modificación
// Regresa : Nada
   Permite dar de alta un registro de la Base de Datos activa
   al momento de hacer la llamada.
// También se puede emplear para modificar un registro
   existente, la diferencia es que en la modificación
// no se pueden cambiar los campos claves.

LOCAL Y := 0
LOCAL X_Pantalla := SAVESCREEN( 0, 0, 24, 79 )
LOCAL X_Color    := SETCOLOR()
LOCAL X_Bloque := { || SETCOLOR( X_Color ), ;
                  RESTSCREEN( 0, 0, 24, 79,  X_Pantalla ),;
                  NIL }

// Crea e inicializa las Variables a utilizar
 FOR Y := 1 TO X
   IF Arreglo[ Y, 3 ] <> "O"
     &("X"+Arreglo[Y,1]):=IF(Mod,&(Fdbf+"-
>"+ALLTRIM(Arreglo[Y,1])),&(ALLTRIM(Arreglo[Y,6])))
     ENDIF
 NEXT

 CLEAR GETS
```

```
// Dibuja la sombra y el marco
SETCOLOR("N/N")
@ 2 + SELECT(), 8 + SELECT() CLEAR TO X + 4 + SELECT(), 64 +;
  SELECT()
SETCOLOR("W+/BG, W+/RG, , , W/G")
@ 1 + SELECT(), 7 + SELECT() CLEAR TO X + 3 + SELECT(), 63 +;
  SELECT()
@ 1 + SELECT(), 7 + SELECT()  TO X + 3 + SELECT(), 63 +;
  SELECT() DOUBLE

Y:= 0
DO WHILE ++Y <= X .AND. Arreglo[ Y, 3 ] $ "CK"
  Variable := "X" + Arreglo[ Y, 1 ]

  DO CASE
  // Modificación de un campo llave modificable
  CASE Mod .AND. Arreglo[ Y, 3 ] = "C"
    Variable := &( Fdbf + "->" + ALLTRIM( Arreglo[ Y,1 ] ) )
    @ 2 + SELECT(), 10 + SELECT() SAY ;
    ALLTRIM( Arreglo[ Y, 2 ] ) + " : "
    @ ROW(), COL() SAY Variable

  // Alta de campo llave no modificable
  // o Modificación de campo llave modificable
  CASE Arreglo[ Y, 3 ] = "C" .OR. Arreglo[ Y, 3 ] = "K"
    REPEAT
      @ 2 + SELECT(), 10 + SELECT() SAY ;
        ALLTRIM( Arreglo[ Y, 2 ] ) + " : ";
        GET &Variable ;
        VALID( &(ALLTRIM( Arreglo[ Y, 4 ] ) ) ) ;
        PICTURE  &(ALLTRIM( Arreglo[ Y, 5 ] ) )
      READ

      IF LASTKEY() = 27  // ESCAPE
        RETURN EVAL( X_Bloque )
      ENDIF

    UNTIL Chk_Clave( Fdbf, &Variable )
  ENDCASE

ENDDO
```

```
// Edición de campos no clave
Z := Y
REPEAT
 Y := Z

 DO WHILE Arreglo[ Y, 3 ] = "N" .AND. Y <= X
  Variable := ("X" + Arreglo[ Y, 1 ])
  @ 2 + Y + SELECT(), 10 + SELECT() SAY ;
      ALLTRIM( Arreglo[ Y, 2 ] ) + " :" ;
      GET &Variable ;
      VALID( &( ALLTRIM( Arreglo[ Y, 4 ] ) ) ) ;
      PICTURE  &( ALLTRIM( Arreglo[ Y, 5 ] ))
  READ
  IF( LASTKEY() = 27, Y--, Y++ )
 ENDDO

 IF Y < Z
  RETURN EVAL( X_Bloque )
 ENDIF

UNTIL MENSAJE( "DATOS CORRECTOS (S/N/A)", "SNA" ) $ 'SA'

IF  LASTKEY() = ASC('A') .OR. LASTKEY() = ASC('A') .OR.;
  LASTKEY() = 27
 RETURN EVAL( X_Bloque )
ENDIF .

Salvar( Fdbf, !Mod )

// Ejecución de las funciones de los Otros campos
DO WHILE Arreglo[ Y, 3 ] = "O" .AND. Y <= X

 &( Arreglo[ Y, 4 ] ) ( &( Arreglo[ Y, 5 ] ) )
```

```
  IF LASTKEY() = 27
   RETURN EVAL( X_Bloque )
  ENDIF

  Y++

 ENDDO

RETURN EVAL( X_Bloque )

FUNCTION Consulta( Fdbf )
// Recibe  : 1.- Alias de la Base de Datos activa
// Regresa : Nada
//  Permite consultar el registro apuntado en ese momento,
//  de la Base de Datos activa.

LOCAL Y := 0
LOCAL X_Pantalla := SAVESCREEN( 0, 0, 24, 79 )
LOCAL X_Color    := SETCOLOR()
LOCAL X_Bloque := { || SETCOLOR( X_Color ), ;
                RESTSCREEN( 0, 0, 24, 79, X_Pantalla),;
                NIL}

 // Dibuja sombra y marco
 SETCOLOR("N/N")
 @ 2 + SELECT(), 8 + SELECT() CLEAR TO X + 4 + SELECT(), 64 +;
   SELECT()
 SETCOLOR("W+/BG, W+/B+, , , W/G")
 @ 1 + SELECT(), 7 + SELECT() CLEAR TO X + 3 + SELECT(), 63 +;
   SELECT()
 @ 1 + SELECT(), 7 + SELECT() TO X + 3 + SELECT(), 63 + ;
   SELECT() DOUBLE

 FOR Y := 1 TO X
```

```
  // Consulta de campos Normales
  IF Arreglo[ Y, 3 ] <> "O"
   @ 1 + Y + SELECT(), 10 + SELECT() SAY ;
    ALLTRIM( Arreglo[ Y, 2 ] ) + " :"
   @ ROW(), COL() SAY &( Fdbf + "->" + ALLTRIM( Arreglo[Y,1]))
  ELSE
   // Ejecución de funciones de los otros campos.
   &( Arreglo[ Y, 4 ] ) ( &( Arreglo[ Y, 5 ] ) )
  ENDIF

 NEXT

 IF Arreglo[ --Y, 3] <> "O"
   WAIT ""
 ENDIF

RETURN EVAL( X_Bloque )
#command REPEAT                 => WHILE .T.
#command UNTIL <exp>            => IF <exp> ; EXIT ; END ; END
```

```
//
// Proyecto  : Sistema ejemplo para Tesis
// Programa  : GRALES.PRG
// Funciones :  Chk_Clave()
//              Mensaje()
//              Salvar()
//              Chk_Datos()
//              Buscar()
//              Ayuda()
//

FUNCTION Chk_Clave( Alias, Key )
// Recibe  :          1.- Base de Datos sobre la cual hará  la búsqueda
//                    2.- Clave a buscar
// Regresa :          Verdadero          => Si encontró la clave
//                    Falso              => Si no la encontró
// Busca el la base de datos especificada la clave
//  correspondiente,
//  Si la encuentra despliega un mensaje de error,
//  en otro caso regresa verdadero.

LOCAL RecNumber

 SELECT (Alias)
 RecNumber := RECNO()
 SEEK Key

 IF FOUND()
  MENSAJE( "C L A V E   E X I S T E N T E", " " )
  GO RecNumber
  RETURN ( .F. )
 ENDIF

 // Regresa el puntero a la posición original
 GO RecNumber

RETURN ( .T. )
```

```
FUNCTION Mensaje( Mensaje, Opciones )
// Recibe   :          1.- Mensaje que se desplegará
//                     2.- Teclas válidas para salir del mensaje
// Regresa  : La tecla presionada al desplegarse el mensaje
// Despliega el mensaje que se le envíe como parámetro y regresa
//  tecla presionada en respuesta al mensaje, la cual sólo puede
//  ser una de las opciones del segundo parámetro.

LOCAL Opcion      := ''
LOCAL P           := (79-LEN(Mensaje))/2
LOCAL X_Pantalla  := SAVESCREEN( 14, 0, 17, 79 )
LOCAL X_Color     := SETCOLOR()

 // Dibuja sombra y marco
 SETCOLOR("N/N")
 @ 15, P - 2 CLEAR TO 17, P + Len ( Mensaje ) + 4
 SETCOLOR("W+/b+, W+/B+")
 @ 14, P - 3 CLEAR TO 16, P + Len ( Mensaje ) + 3
 @ 14, P - 3 TO 16, P + Len ( Mensaje ) + 3 DOUBLE

 Opciones := UPPER(Opciones) + LOWER(Opciones)
 @ 15, P    SAY Mensaje  GET Opcion PICTURE "A" ;
                  VALID( IF (Opciones = " ", .T., ;
                        Opcion $ Opciones ) )
 READ
 SETCOLOR( X_Color )
 RESTSCREEN( 14, 0, 17, 79, X_Pantalla )

RETURN UPPER( Opcion )

FUNCTION Salvar( Fdbf, Flag )
// Recibe :     1.- Base de Datos sobre la cual se guardaron los
//                  datos
//              2.- Modificación o Alta (Verdadero o Falso)
// Regresa : Nada
// Guarda los datos que se leyeron en la base de datos enviada
//  como primer parámetro. Si el segundo parámetro es verdadero
//  se refiere a dar de alta, el otro caso sólo son cambios.
```

```
 SELECT &Fdbf

 IF Flag
  APPEND BLANK
 ENDIF

 FOR Y := 1 TO X

  IF Arreglo[ Y, 3 ] <> "O"
   REPLACE &( Fdbf + "->" + Arreglo[ Y, 1 ] ) ;
    WITH &("X" + Arreglo[ Y, 1 ] )
  ENDIF

 NEXT

RETURN NIL

FUNCTION Chk_Datos( Var, Arr )
// Recibe :    1.- Clave a buscar
//             2.- Arreglo de Valores posibles
// Regresa :   Verdadero     => Si encontró la clave
//             Falso         => Si no la encontró
// Busca en la tabla de valores enviada como segundo parámetro,
//  la clave enviada como primer parámetro
//  Si no la encuentra despliega una tabla con los valores
//  válidos.

LOCAL X_Pantalla := SAVESCREEN( 0, 0, 24, 79 )
LOCAL Resp       := ""
LOCAL Max        := 0

 FOR W:= 1 TO LEN( Arr )
  // Carga las respuestas posibles a la variable Resp
  Resp := Resp + SUBSTR( Arr[W], 1, 1 )
  // Determina cual es la respuesta con longitud más
  //  grande, para dibujar el marco con respecto a ésta.
  Max  := IF( LEN( Arr[W] ) > Max, LEN( Arr[W] ), Max )
 NEXT
```

```
// Checa si Var, esta dentro de las posibles válidos
IF Var $ Resp
  RETURN .T.
ENDIF

// Dibuja sombra y marco
Color := SETCOLOR()
SETCOLOR( "N/N" )
@ ROW(), COL() CLEAR TO ROW()+W, COL()+Max+1
SETCOLOR( Color )
@ ROW()-1, COL()-1 TO ROW()+W-1, COL()+Max DOUBLE

// Despliega las posibles respuestas
Var := SUBSTR(Resp, ;
     ACHOICE( ROW(), COL(), ROW()+W, COL()+Max-1, Arr), ;
     1 )
RESTSCREEN( 0, 0, 24, 79, X_Pantalla )

RETURN .F.

FUNCTION Buscar( Alias, Key )
// Recibe :    1.- Base de datos sobre la cual se hará la búsqueda
//             2.- Clave a buscar
// Regresa :   Verdadero    => Si encontró la clave
//             Falso        => Si no la encontró
// Busca el la Base de datos correspondiente la clave
//  especificada.
//  Si no la encuentra manda llamar a la función hojear, la cual
//  crear un objeto para esa base de Datos.
//  En otro caso regresa verdadero.

LOCAL X_Alias   := SELECT()
LOCAL X_Pantalla := SAVESCREEN( 0, 0, 24, 79 )
LOCAL X_Color   := SETCOLOR()
LOCAL X_Retorno := .T.

 SELECT &Alias
 SEEK Key
```

```
  // Si no encontró la clave
  IF !FOUND()
    CLEAR GETS
    GO TOP
    Hojear( Alias, ROW(), COL(), ROW()+6, COL()+35, 2 )
    Key := &( Alias + "->" + Fieldname(1) )
    X_Retorno := .F.
  ENDIF

  RESTSCREEN( 0, 0, 24, 9, X_Pantalla )
  SETCOLOR( X_Color )
  SELECT  (X_ALIAS)

RETURN ( X_Retorno )

FUNCTION Ayuda( Modo )
// Recibe  : Modo
// Regresa : Nada
// Despliega el mensaje de ayuda dependiendo del modo del Browse
LOCAL X_Pantalla := SAVESCREEN( 2, 15, 23, 71 )
LOCAL X_Color    := SETCOLOR()
  // Dibuja marco y sombra
  SETCOLOR("N/")
  @ 3, 16 CLEAR TO IF( Modo <> 3, 22, 19 ), 71
  SETCOLOR("W+/B+, W+/B+")
  @ 2, 15 CLEAR TO IF( Modo <> 3, 21, 18 ), 70
  @ 2, 15 TO IF(Modo <> 3, 21, 18 ), 770 DOUBLE
  @ 3, 30 SAY "TECLAS VALIDAS:"
  @ 4, 23 SAY "Flecha Abajo              => Cursor hacia Abajo"
  @ 5, 23 SAY "Flecha Arriba             => Cursor hacia Arriba"
  @ 6, 23 SAY "Flecha Izquierda          => Cursor hacia Izquierda"
  @ 7, 23 SAY "Flecha Derecha            => Cursor hacia Derecha"
  @ 8, 23 SAY "P gina Abajo              => Mover una página Abajo"
  @ 9, 23 SAY "P gina Arriba             => Mover una página Arriba"
  @ 10, 23 SAY "Ctrl+P gina Abajo        => Ir al primer registro"
  @ 11, 23 SAY "Ctrl+P gina Arriba       => Ir al último registro"
  @ 12, 23 SAY "Inicio                   => Ir a la primera columna visible"
  @ 13, 23 SAY "Fin                      => Ir a la última columna visible"
  @ 14, 23 SAY "Ctrl+Inicio              => Ir a la primera columna"
  @ 15, 23 SAY "Ctrl+Fin                 => Ir a la última columna"
  @ 16, 23 SAY "Escape                   => REGRESAR a lo Anterior"
```

```
DO CASE
 CASE Modo = 1
  @ 17, 23 SAY "Enter => CONSULTAR registro activo"
 CASE Modo = 2
  @ 17, 23 SAY "Enter => ELEGIR y REGRESAR a lo Anterior"
 CASE Modo = 3
  @ 17, 23 SAY "Enter => REGRESAR a lo Anterior"
ENDCASE

IF Modo <> 3
 @ 18, 23 SAY "Insertar          => Dar de ALTA Nuevo Registro"
 @ 19, 23 SAY "Retroceso         => MODIFICAR registro activo"
 @ 20, 23 SAY "Borrar            => BORRAR el registro activo"
ENDIF

    WAIT ""

   SETCOLOR( X_Color )
   RESTSCREEN ( 2, 15, 23, 1, X_Pantalla )
   RETURN NIL
```

CONCLUSIONES

De acuerdo a lo anteriormente investigado y expuesto acerca de la Programación Orientada a Objetos, tenemos las siguientes conclusiones acerca de este tema:

- La programación Orientada a objetos es una metodología para crear Software bien estructurado, de forma "natural", similar a la forma de estructurar objetos en el mundo real. Recordemos que todo en el mundo real son objetos, y la Programación Orientada a Objetos toma esta idea para desarrollar programas para computadoras.

- Permite la creación de código reutilizable, es decir, que se puede utilizar en varias aplicaciones, sin grandes cambios.

  Esto es debido a que un objeto tiene bien definidos cuales son sus métodos o comportamiento, los cuales son únicos para este objeto.

- La Programación Orientada a Objetos a pesar de sus 25 años de creación, ha salido a la luz sólo recientemente gracias a que ha aumentado la velocidad del Hardware y han salido más lenguajes de programación que utilizan este tipo de programación.

- La Programación Orientada a Objetos permite crear objetos simulando el mundo real, los cuales pueden combinar atributos y comportamiento en un sólo paquete. A diferencia de la Programación Estructurada la cual sólo utiliza procedimientos sin importar a que objeto o entidad se aplicará.

- La diferencia fundamental entre la Programación Estructurada y la Programación Orientada a Objetos, es que ésta utiliza mensajes entre los objetos, en lugar de llamadas a subrutina o procedimientos. Los mensajes que recibirá cada objeto están claramente definidos al momento de definir el mismo objeto, por lo cual, este objeto sólo puede recibir los mensajes que le fueron asignados.

- Para que un lenguaje de Programación se considere orientado a Objetos debe soportar las características de Herencia, Polimorfismo y Encapsulación.

- Los conceptos de Programación Orientada a Objetos fueron introducidos por primera a través del lenguaje de Programación SIMULA en 1967 en Noruega.

- A través de la reutilización de código se eleva la productividad en el desarrollo de nuevas aplicaciones, y se evita la reprogramación para futuras aplicaciones que utilicen los mismos objetos.

- Los beneficios de la Programación Orientada a Objetos también se extienden al área de Programación de Sistemas Operativos, lo cual permitirá en un futuro cercano la interoperatividad entre las diferentes plataformas Hardware.

- La aplicación del enfoque Orientado a Objetos hacia el desarrollo del sistema de información presentado, permitió crear aplicaciones independientes de estructuras de Datos rígidas dentro de los programas, además de que permite cierto tipo de Polimorfismo de Subtipos. Finalmente, se llegó al fin primordial de la Programación Orientada a Objetos; crear código reutilizable para el rápido desarrollo de aplicaciones. El programa presentado puede ser utilizado para realizar distintas aplicaciones y con algunas modificaciones quedará más completo; agregando algún tipo de reporte, operaciones especiales o procesos especiales.

- En términos generales la Programación Orientada a Objetos; agrega Calidad al Software, lo cual crea aplicaciones libres de errores, permite un rápido desarrollo, permitiendo completar totalmente proyectos de software y crea código reutilizable, para utilizarse en cualquier aplicación futura.

COMENTARIOS FINALES

- El preparar a un programador junior en esta Tecnología y que comience a ser productivo bajo la Programación Orientada a Objetos, puede llevar aproximadamente 6 meses o más.

- Puede ser difícil en un principio adentrarse en esta forma de pensar para resolver problemas, sin embargo, al dominar la Programación Estructurada, el paso hacia la Programación Orientada a Objetos es sólo dejar de pensar en procedimientos y comenzar a pensar en objetos y métodos principalmente.

- Muchos autores de este tema sugieren que para programar 100% en objetos es necesario utilizar un lenguaje puro, en vez de alguno híbrido. Sin embargo esto no es ningún obstáculo para comenzar, incluso se puede simular programar en objetos en lenguaje C estándar.

- La Programación Orientada a Objetos tendrá todavía cambios en los siguientes años, debido a que algunos de los conceptos importantes de esta metodología o las formas de implementar estos conceptos están todavía en estudio, sin embargo los conceptos básicos no cambiarán de todo.

- La Programación Orientada a Objetos permite crear Software bien estructurado, emulando objetos del mundo real. Sin embargo, el Software no se crea solo, detrás de cada programa existe una lógica, la cual es estructurada por algún programador y este hecho no cambia, independientemente de la metodología que se utilice.

GLOSARIO

Abstracción

Ocultar los detalles irrelevantes para la comprensión y manipulación del tipo de dato.

Agregación

La Agregación representa la relación "tener a" o "consistir de". Es la forma natural de conceptualizar a los objetos, creados a partir de otros objetos.

Asociación

La Asociación es el uso directo de los servicios de un objeto por otro. Esta relación se basa en la interacción entre dos objetos independientes uno de otro, los cuales se utilizan mutuamente para lograr sus objetivos.

Clase

Una plantilla utilizada para crear nuevos objetos. Una clase incluye tanto datos como métodos.

Compilación

Proceso mediante el cual se "traduce" de un lenguaje de alto nivel (entendible al programador) a lenguaje máquina.

Descomposición Funcional

Dividir un programa sistemáticamente en componentes, los cuales a su vez se subdividen en subcomponentes, hasta el nivel de subrutina.

Herencia

La habilidad de pasar las propiedades de clase de una clase a otra.

Herencia Múltiple

La habilidad de heredar de padres múltiples.

Encapsulación

La representación de un objeto y sus datos en unidades discretas. El aisamiento de un objeto (y sus datos) de otros objetos.

Información Oculta

Técnica de hacer los detalles internos de un módulo inaccesibles a otros módulos, protegiendo así el módulo.

Ingeniería de Software

Aplicación práctica del conocimiento científico en el diseño y construcción de programas para computadoras y la documentación asociada requerida para desarrollarlos, operarlos y mantenerlos.

Instancia

Una implementación de la planilla de una clase. Un objeto es una instancia de una clase, el cual contiene la implementación de la interfase de datos.

Interoperatividad

Habilidad de que un solo Software pueda funcionar en diferentes plataformas Hardware.

Jerarquía de clase

Estructura de árbol que representa las relaciones entre grupos de clases.

Lenguaje de Programación

Conjunto de caracteres y reglas bien definidas sobre cuya base es posible escribir un programa para computadoras.

Lenguaje orientado a objetos

Es un lenguaje de programación que utiliza objetos y que es capaz de tener herencia de clase.

Liga Dinámica

Asocia el significado de una función al momento de su activación.

Mensaje

El mecanismo mediante el cual se comunican los objetos.

Método

Una operación que efectúa acceso a datos.

Objeto

Es una entidad que tiene unos atributos particulares, los datos y formas de operar sobre ellos, los procedimientos.

Plataforma Hardware

Denominación de los diferentes tipos de computadoras o ambientes de sistemas operativos, los cuales contienen características que los diferencian de las demás plataformas.

Polimorfismo

La habilidad de enviar el mismo mensaje a objetos de diferentes clases.

Portabilidad

Definición de lenguaje independiente en cualquier plataforma, la misma semántica del lenguaje puede implementarse en cualquier plataforma.

Programación Estructurada

Metodología de Programación la cual se apoya en la descomposición funcional considerando una disciplina y estilo de programación.

Programación Modular

Metodología de Programación que consiste en dividir un problema grande en pequeños componentes, que puedan ser construidos independientemente, luego combinar éstos hasta formar un sistema completo.

Protocolo

Los mensajes a los que responde un objeto.

Reusabilidad

Habilidad de producir componentes que puedan utilizarse en diferentes aplicaciones.

Sistema

Conjunto de elementos interrelacionados con un objetivo común.

Sistema de Información

Conjunto de todos los procedimientos y dispositivos implicados en el proceso, almacenamiento y distribución de la información en una organización.

Sobrecarga

Posibilidad de definir varias funciones con el mismo nombre

Software

Conjunto de programas que cargados en una computadora hacen que ésta funcione.

Subclase

Una clase que tiene todas las propiedades de su clase padre

Superclase

Una clase padre de una subclase.

Tipo de dato abstracto

Tipo de dato definido por el programador y no forma parte del lenguaje de programación.

Valor de retorno

Respuesta del mensaje enviado hacia un objeto por medio de otro objeto que envió el mensaje.

BIBLIOGRAFÍA

1. Object-Oriented Tecnology: A manager guide
   David A. Taylor
   Editorial Adison-Wesley
   Estados Unidos, 1992

2. A book of Object-Oriented Knoledge
   Brian Henderson-Sellers
   Editorial Prentice Hall. Serie Orientada a Objetos.
   Australia , 1992

3. Clipper 5. Referencia rápida
   Francisco Marín, Antonio Quiróz y Antonio Torres
   Editorial Macrobit Editores, S.A. de C.V.
   México, 1991

4. Curso de programación C++, Programación Orientada a Objetos
   Francisco Javier Ceballos
   Editorial Addison-Wesley Iberoamericana, S.A. de C.V.
   Estados Unidos, 1993

5. CLIPPER 5
   José Antonio Ramalho
   Editorial Mc Graw Hill
   Colombia 1992

6. CLIPPER 5.2 a su alcance
   José Javier García-Badell
   Editorial Mc Graw Hill
   España 1994

7. OBJECT ORIENTED DESIGN
   Grady Booch
   Editorial The Benjamin/Cummings
   Segunda Edición

8. Revistas

Journal of Object-Oriented programming
Vol.6 #4. Julio-Agosto de 1993.
SIGS Publication

Personal Computing México
Ann Steffora
Editorial Servicios Editoriales Sayrols, S.A. de C.V.
Agosto de 1993

PC/TIPS #42
Programación Orientada a Objetos en Clipper
Alejandro Reskala Ruiz
Julio 1991

PC/TIPS BYTE México #52
Programación Orientada a Objetos en un ambiente puro
Abel Archundia Pineda y Jorge Muñoz Márquez
Mayo de 1992

PC/TIPS BYTE México #53
OOP vía DDE
Kenvin Kornfeld y Kevin Gilhooly
Junio de 1992

PC/TIPS BYTE México #58
Bruce D. Schatzman
Noviembre de 1992

PC WORD (España) #75
Artículo: Sistemas Operativos orientados a objetos.
Editorial IDG Communications
Marzo de 1992

PC WORD (España) #77
Artículo: Introducción a la Ingeniería de Software
Editorial IDG Communications
Mayo de 1992

PC WORD (España) #83
Artículo: La Evolución del desarrollo de aplicaciones
Editorial IDG Communications
Diciembre de 1992

Soluciones Avanzadas #4
Articulos: Elementos de la Programación Orientada a Objetos
An lisis y Diseño Orientados a Objetos
Programación Orientada a Objetos desde el punto de vista de un Ingeniero. (Charla con Frederic Deramar)
Editorial Xview, S.A. de C.V.
Julio-Agosto de 1993

Soluciones Avanzadas #5
Articulos: Abstracción de Datos en Lenguajes Orientados a Objetos...
El ocaso de los paradigmas de Programación.
Editorial Xview, S.A. de C.V.
Septiembre-Octubre de 1993

Soluciones Avanzadas #6
Articulos: Nuevas Tecnologías para el Desarrollo de aplicaciones.
Análisis y Diseño Orientados a Objetos
Editorial Xview, S.A. de C.V.
Noviembre-Diciembre de 1993

Soluciones Avanzadas #8
Articulos: Principio Abierto Cerrado: C++ vs. Eiffel
Polimorfismo en los Lenguajes de Programación
Editorial Xview, S.A. de C.V.
Marzo-Abril de 1994

Soluciones Avanzadas #10
Articulos: Análisis Orientado a Objetos mediante la técnica de Modelado de Objetos OMT
Un panorama de las Bases de Datos Orientadas a Objetos
Editorial Xview, S.A. de C.V.
Junio de 1994.