UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

LA TECNOLOGIA ORIENTADA A OBJETOS EN SISTEMAS DE MANEJO DE BASES DE DATOS DE TERCERA GENERACION

SEMINARIO DE INVESTIGACION INFORMATICA
QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN INFORMATICA

P R E S E N T A:

MARIA BEATRIZ PONCE ARANDA

ASESOR DEL SEMINARIO:

M. en A. LUIS EDUARDO LOPEZ CASTRO

MEXICO, D. F. 1996

TESIS CON
FALLA DE ORIGEN

A Queridos Padres:

A mi Papá:

Armando Ponce Huerta

Por ser mi mejor guía en mi camino, porque en toda mi existencia me ha brindado su apoyo, su cariño y comprensión, por darme el valioso ejemplo de la honradez y la constante lucha; de quién siento un gran orgullo.

Que Dios te bendiga siempre.

A mi Mamá:

Esperanza Aranda de Ponce

Es una gran tristeza que no estés aquí para poder compartir contigo lo que siempre esperaste de mi. Bendice mi camino desde donde te encuentres.

A mi esposo

Sergio

Gracias por tu amor y compañía que siempre ha existido. Por tu "apoyo y ejemplo" para llegar a realizarme como ser humano y profesionista.

A mi preciosa hija

Ana Lilia

Por la alegría, ternura y cariño que ha traído a mi vida desde su llegada.

A mis hermanas:

Ma. Eugenia, Guillermina, Yolanda y Esperanza.

Mi infinito cariño y respeto porque son un gran ejemplo como personas, por todo lo que me han dado en la vida y por haberme motivado siempre para seguir adelante.

A mis Tías:

Raquel y Magdalena Ponce Huerta

Por el amor y muy positivos consejos que siempre han dado a mi vida.

A mis Primas:

Lety y Paty

Porque con su cariño y entusiasmo me dieron el impulso para alcanzar mis metas.

A Lulú y Eduardo

Que con su apoyo y comprensión me alentaron en mi vida estudiantil para lograr este gran sueño.

A Rosita

Por su amistad, apoyo y respeto recíproco.

A todas aquéllas personas, que directa o indirectamente contribuyeron al logro de mi propósito.

A mi Asesor de Seminario y Profesor de la Facultad

M. en A. Luis Eduardo López Castro

Que nunca escatimó tiempo, ni conocimientos para hacer posible la realización del presente trabajo, y que es un excelente ejemplo de sencillez y profesionalismo.

Doy Gracias a Dios por su infinita bondad de concederme el anhelo de lograr esta meta.

A la Universidad Nacional Autónoma de México

Que cada año abre las puertas a la juventud mexicana con la finalidad de prepararnos y hacernos útiles a la sociedad.

CONTENIDO
I

PREFACIO

Es evidente la velocidad a la que avanza la tecnología de hardware de computadoras. Día a día se logran avances que conducen a la construcción de computadoras más veloces, más compactas y más baratas. Sin embargo, en el aspecto de software no parece haber un desarrollo similar. Mientras los costos de hardware han disminuido continuamente, los de software han hecho lo contrario. La construcción de software no es una tarea fácil por los problemas de calidad que presenta, y en muchas ocasiones los proyectos de programación sobregiran los presupuestos de tiempo y dinero. Esta condición de la industria es lo que generalmente se conoce como la "Crisis del Software".

Durante la década de 1970 se realizaron importantes avances, tanto en la industria como en el mundo académico, para caracterizar mejor los productos de software y establecer una base empresarial para su desarrollo a gran escala. Se les dio importancia a los conceptos de la programación estructurada, se crearon diversas metodologías estructuradas efectivas para el análisis y diseño y se avanzó notablemente en materia de métodos formales. Estos esfuerzos condujeron a la posterior consolidación de la Ingeniería de Software.

Durante la década de 1980, continúo el avance en métodos formales, se desarrollaron las métricas de software, las herramientas CASE y el empleo de prototipos. Además un nuevo paradigma en la práctica del desarrollo de software es el paradigma de los objetos..

La esencia de programación orientada a objetos es el ocultamiento o encapsulamiento del estado interno de las entidades y la especificación de sus propiedades interactivas por medio de una interfaz de operaciones (los eventos en que las entidades pueden participar). Algunas publicaciones y un intensivo desarrollo de herramientas CASE han venido a consolidar las "Metodologías orientadas a objetos". El gran interés que ha despertado esta nueva tecnología se explica por su enorme potencial en el incremento de la productividad y la calidad.

La programación orientada a objetos fue uno de los primeros sistemas de programación en que se reconoció que el cambio y la evolución de un programa no solo son inevitables, sino deseables. La programación orientada a objetos busca minimizar el impacto del cambio a través de la técnica de encapsulado. También apoya la reutilización de código ya existente y hace que esta sea práctica.

En el primer capítulo es importante iniciar con una perspectiva global de la estructura de la programación orientada a objetos. **¿Qué es la Programación Orientada a Objetos?** Si uno se restringe al significado exacto de la programación orientada a objetos, esta es la programación por medio del envío de mensajes a objetos de caracter desconocido. Dichos objetos pueden ser encontrados en arreglos o en una colección como es un escritorio.

Todos los objetos en la colección comparten ciertas características (como el conocimiento de la posición en la pantalla y la habilidad de ser reubicado, activado y desactivado). Desde la perspectva del programador, la selección de un objeto de la colección no le proporciona información sobre el tipo exacto de el. Dado que el

tipo de un objeto no es conocido, cuando el mensaje es enviado, la respuesta exacta del objeto no puede ser predicha. De nuevo enviar mensajes a objetos de tipo desconocido es una técnica poderosa de programación y eso es lo que realmente significa programación orientada a objetos. Además se da un enfoque general de los conceptos básicos referentes a esta tecnología.

En el segundo capítulo se inicia con la explicación del principio de organización de una base de datos relacional que es la tabla, posteriormente se especifican varios tipos de datos que pueden ser almacenados en una base de datos basada en SQL. Además se da un conocimiento general de los tipos estructurados y BLOBS (Imágenes, Sonidos-Multimedia y por último se explica ampliamente el lenguaje de consulta SQL.

En el tercer capítulo se presenta el almacenamiento de objetos en una base de datos en el cuál los objetos son los elementos que una base de datos orientada a objetos almacena, modifica y recupera por orden del programa de la aplicación. Estos objetos pueden ser tan simples como números y cadenas o tan complejos como las especificaciones completas de un circuito electrónico. Tienen cabida también los tipos múltiples de datos, incluyendo gráficos, sonidos y video. Una de las principales ventajas de las bases de datos orientadas a objetos tienen que sobre las bases de datos tradicionales es su capacidad para representar como objeto cualquier entidad del mundo real. La alternativa, es comprimir los datos en tablas inconexas. Las diferencias fundamentales entre tablas y objetos de bases de datos, es que un objeto puede contener otros objetos en cualquier nivel de anidamiento, proporcionando así una tremenda flexibilidad al definir nuevos tipos de objetos. Esto quiere decir que los objetos hacen que sea más fácil la escritura de las aplicaciones, porque todos los datos de una entidad determinada están localizados en un lugar.

Por último en el **cuarto capítulo** se dan ejemplos de algunas Bases de Datos Orientadas a Objetos que ya se encuentran a la venta en el mercado de la computación como son : POSTGRES, IRIS, VBASE, SIM, GEMSTONE Y PICQUERY. Aquí se representan los puntos más importantes de cada paquete para hacer un estudio comparativo.

1. TECNOLOGIA ORIENTADA A OBJETOS

1.0 INTRODUCCION

Es importante aprender la programación orientada a objetos debido a que en la actualidad para construir un software complejo ya no es suficiente pegar de cualquier forma secuencias de instrucciones de máquina, proposiciones en lenguaje de alto nivel o incluso conjuntos de procedimientos y módulos, con la esperanza de encontrar la forma de recorrer el laberinto de nombres de variable, de procedimiento y de archivo, para armar un programa robusto. Los complejos programas actuales requieren de técnicas prácticas de construcción junto con lineamientos para crear una sólida estructura de programa que sea fácil de desentrañar, comprender y modificar.

La programación estructurada ha proporcionado a los programadores una herramienta de poder inmesurable. En la actualidad, la mayoría de los programadores ha estudiado las técnicas de programación estructurada y toma como un hecho los principios arquitectónicos introducidos por la programación estructurada. La programación estructurada puede igualarse al descubrimiento del arco: que permitió a los ingenieros y arquitectos de software realizar construcciones que no solo fueron más fuertes y estables, sino que se realizaron con menos material. les permitió hacer más con menos.

Lograr más con menos es posible a través del descubrimiento de principios generales y de la aplicación de esos principios para la construcción de sistemas. Los principios deben descubrirse, articularse y probarse primero. Si un principio general sobrevive los rigores de las pruebas y demuestra ser una herramienta útil, entonces resulta tonto no aplicarlo en la práctica. La programación estructurada presenta un conjunto de principios para la comprensión de los sistemas de software que son el resultado de años de observación y estudio de las técnicas de construcción de programas.

La programación orientada a objetos es otro paso evolutivo en el diseño y construcción de software.

Los programadores acostumbrados a pensar en términos de procedimientos con frecuencia se sienten desorientados al tratar de comprender la finalidad de los objetos y la estructura de los programas orientados a objetos. Escuchan una y otra vez que la programación orientada a objetos requiere una forma de pensar completamente nueva: olviden todo lo que ya saben. Nada podría estar más alejado de la verdad. Sin importar lo que alguien diga, mientras más práctica tenga usted en los principios de buen diseño y codificación estructurados, le será más fácil realizar la transición a objetos.

La programación orientada a objetos simplemente formaliza prácticas que los buenos programadores ya utilizan con los lenguajes estructurados. Aún con los objetos, la codificación en esencia se traduce en la construcción de

procedimientos o métodos. Es simplemente el empacado de los componentes de programa lo que resulta diferente. La programación orientada a objetos requiere que usted describa en forma explícita las relaciones entre los procedimientos y datos compartidos.
Los lenguajes orientados a objetos simplemente formalizan la estructura y proporcionan una forma para que el compilador asegure que el código no viola las restricciones de acceso que usted especifica.
No hay nada nuevo en la programación orientada a objetos y sin embargo todo es nuevo. Esta paradoja impide a los programadores saber cómo y cuando utilizar la programación orientada a objetos. Hoy en día, más que nunca, los principios de la programación estructurada se mantienen firmes. Lo que resulta nuevo es un ligero giro mental, un cambio en perspectiva que nos permite ver una estructura que se encuentra en la mayoría de los programas basados en procedimientos hechos con buen oficio.
Las personas que inventaron y desarrollaron la programación orientada a objetos no empezaron con las frases básicas y las características del lenguaje, y pasaron después a hacer una práctica de programación con base en esas frases y características. Eran programadores con mucha experiencia que utilizaron, junto con su conocimiento, para buscar formas de simplificar el proceso de programación y acabar con su frustración por las herramientas y técnicas existentes. En resumen, empezaron con un concepto general y una motivación: hacer que los programas fueran más fáciles de construir y de comprender.
La programación orientada a objetos es una técnica de estructuración. En la programación orientada a objetos, los objetos son los principales elementos de la construcción. No obstante, el simple entendimiento de lo que es un objeto o los objetos en un programa no significa necesariamente que se esté dentro de lo que es la programación orientada a objetos.
En la programación procedural es posible construir un programa a partir de los elementos del procedimiento y entonces violar los principios de la construcción. El simple conocimiento de lo que es un procedimiento o de como construirlo no garantiza la solidez de un programa fuerte, libre de errores, fácil de usar, fácil de leer y fácil de modificar. Esto mismo es válido para la programación orientada a objetos.

1.1 CLASES Y OBJETOS

Una clase es una familia de objetos de un tipo particular. Los objetos que pertenecen a una clase son llamados instancias, por ello una clase puede tener una o varias instancias, pero una instancia tiene sólo una clase, esto se conoce como herencia simple.
Una clase será a la vez un módulo y un tipo. Como módulo la clase encapsula (encierra) un número de facilidades o recursos que ofrecerá a otras clases (sus clientes). Como tipo describe un conjunto de objetos o instancias que existirán en tiempo de ejecución. La conexión entre éstos dos enfoques es muy simple: los

recursos ofrecidos por el módulo son precisamente las operaciones que llevarán a cabo las instancias del tipo.
La POO consiste en definir tipos o clases de objetos que cumplen determinadas propiedades y que pueden llevar a cabo una determinada funcionalidad. Los detalles internos de cómo se logra esa funcionalidad deben quedar ocultos (encapsulados). De éste modo los diferentes niveles de complejidad de un problema irán quedando ocultos en los objetos que se utiiizan para resolverlo.

Podemos considerar que programar bajo el paradigma de objetos consiste en

- programar clases
- crear objetos a partir de las clases
- enviar mensajes a los objetos (llamar a funciones de las clases a través de los objetos).

El programador puede situarse en dos puntos de vista con respecto a una clase:

- Como cliente o heredero de la clase. En este caso humildemente y de buen corazón confía en que la clase que va a utilizar, o de la cual va a heredar, sea robusta y este bien instrumentada. Es decir, supone que ésta no le generara errores si el cumple los requisitos que la clase exige para su uso.
- Como instrumentador de la clase. En éste caso autosuficientemente considera que desarrollara una clase correcta, que no le traerá errores a sus usuarios si estos cumplen disciplinadamente con los requisitos de utilización de la misma.

Una clase por lo general juega estos dos roles : cliente de unas clases y servidor de otras clases estableciéndose durante la ejecución un proceso en espiral que tiene dos extremos:

- Cuando tenemos clases ofrecidas en una biblioteca construida por el propio constructor del compilador (incluso algunas de ellas inmersas dentro del propio lenguaje como las de los tipos básicos y sus operaciones). Para el programador éstas clases son servidores totales (aunque por supuesto internamente éstas clases pueden utilizarse entre sí). En este caso la confianza debe ser máxima (aunque no absoluta porque ningún compilador esta excento de fallas).
- Cuando la clase describe al objeto raíz (objeto cuya creación desencadenará la creación y envío de mensajes a otros objetos).

Un lenguaje orientado a objetos debe permitir agrupar en una misma definición (clase) características de los datos (que determinarán al objeto en tiempo de ejecución y las operaciones que manipulen esos datos. Se trata ahora de describir entidades (objetos) que encierren sus propios datos y que las operaciones que manipulen esos datos sólo puedan ser invocadas a través de los propios objetos.
Cada clase por lo general describe dos grandes grupos de operaciones. Operaciones de **acceso**, que dan información sobre un objeto (y que por lo

general no transformarán al mismo) y operaciones de **transformación** que provocan cambios en el objeto. Las operaciones que dan información sobre el objeto caracterizan lo que se denomina **estado del objeto** y las operaciones de transformación cambian este estado.
POO es un nuevo método de programación que rompe la separación tradicional entre código y datos; es fácil de aprender y puede hacer a sus programas más fáciles de escribir y mantener. Muchas son las propiedades que hacen muy útil a POO, pero seguramente éstas son las más notables: reutilización de código, herencia y encapsulamiento.

LAS PROPIEDADES DE LA POO

Programación orientada a objetos es un conjunto de conceptos de programación que persiguen un objetivo común: creación de módulos reutilizables que faciliten la construcción de grandes programas, merced a la escritura de códigos compactos y fáciles de expandir.
El concepto fundamental de POO es el objeto. Un objeto es una entidad que contiene los datos y los procedimientos y/o funciones que manipulan esos datos. El objeto es similar al registro, pero al contrario que este, que es una estructura de diferentes tipos de datos, un objeto es una estructura de datos y código. Otros conceptos fundamentales de la programación orientada son:

- Tipos objetos, que definen objetos particulares.
- Encapsulamiento, qué enlaza o liga los datos con los códigos que los manipulan (métodos).
- Polimorfismo, la posibilidad de que los diversos objetos actúen de modo diferente en respuesta a una llamada.
- Expansibilidad de código, que permite a unidades ya compiladas ser utilizadas como base para la creación y uso de nuevos objetos que eran desconocidos en tiempo de compilación.

OBJETOS

Los objetos son las unidades estructurales primarias de la programación orientada a objetos. Dar nombres específicos a los datos y elementos procedurales que se encuentran dentro de un objeto ayuda a aclarar la representación y finalidad de dicho objeto.
Podemos representar un punto como un objeto. Los puntos resultan ser objetos muy útiles en sistemas de manejo por ventanas, programas de diseño auxiliado por computadora y herramientas de diseño. Un objeto de punto real proporcionaría muchos mas procedimientos en la interface pública, pero la representación simplificada dará un mejor entendimiento de la estructura y el comportamiento de un objeto típico. Un ejemplo de creación un objeto es el siguiente, utilizando el lenguaje C++ :

```
class Punto
{
   int x;
   int y;
public:
    void setXY (int a, int b)
    {
        x = a;
        y = b;
    }
    int getX  () { return x; }
    int getY  () { return y; }
};
```

Este código es una pantalla que describe los datos internos y los elementos de procedimientos de un objeto Punto. En esta implementación, los objetos punto tienen dos elementos de datos que representan las posiciones x e y en un plano de dos dimensiones. Solo un procedimiento, setXY, puede modificar estos valores de los datos. Los otros dos procedimientos, getX y getY, solo se utilizan para devolver los valores respectivos a objetos y procedimientos que efectúa en alguna solicitud.

Los datos y procedimientos están encapsulados en una capa protectora a fin de prevenir modificaciones indeseadas. Los objetos también recuerdan cosas, esto significa que los objetos tienen un estado. El estado de un objeto es recordado por sus datos componentes. Además, los objetos pueden procesar información, aunque no es mucho el procesamiento que ocurre en el interior del objeto. Los procedimientos solamente establecen los valores de los datos privados y regresan estos valores cuando se solicita. Sin embargo, los objetos reales por lo general tienen una capacidad de procesamiento más compleja.

Lo último a considerar acerca de los objetos, y que particularmente los distinguen de los procedimientos, es que los objetos tienen un ciclo de vida. Es posible crearlos y destruirlos. Siempre y cuando un objeto este vivo y activo, se puede realizar el acceso a cualquiera de sus elementos públicos. Después una vez destruido, todas sus memorias son irrecuperables y no será posible comunicarse con él.

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. Dicho de otra forma, un objeto es una entidad que tiene unos atributos particulares, los datos y una forma de operar sobre ellos, los procedimientos. Los datos que pueden pertenecer a un objeto son, por una parte todos aquellos del tipo convencional (constantes, variables, arrays, strings y registros), y por otra parte las funciones o procedimientos que operan sobre ellos.

Como su nombre implica, la programación orientada a objetos se basa fuertemente en el concepto de objeto. En nuestras vidas diarias estamos familiarizados con cualquier tipo de objeto (televisores, lámparas, cuadernos, etc.)

Pero cuando encendemos la televisión, no distinguimos entre sus elementos físicos (selector de canal, tubo de imagen, antena) y su comportamiento (proporcionar imagen y sonido). Simplemente la encendemos y seleccionamos un canal.
Al igual que la televisión, los objetos hacen que los programas sean un reflejo más fiel de la forma en que tratamos con el mundo real. En POO, los datos y los procedimientos se combinan en objetos. Un objeto contiene las características de una entidad (sus datos) y su comportamiento (sus procedimientos). Combinando estas características y comportamientos, un objeto conoce cualquier cosa que necesite para hacer su trabajo.
Para entender los objetos, es de gran utilidad pensar en términos de metáforas. Un aeroplano se puede describir en términos físicos: el numero de pasajeros que puede albergar, el empuje que genera, su coeficiente de resistencia al avance, etc. Como alternativa, un aeroplano se puede describir en términos funcionales despega, asciende y desciende, gira y aterriza, etc...Ni la descripción física ni la funcionalidad por separado captan la esencia de lo que es un aeroplano; son necesarias ambas descripciones. En POO, las características (datos) y el comportamiento (procedimientos) se combinan en una entidad única llamada objeto.

CLASES

Los mecanismos clásicos de la programación orientada a objetos son Objetos, Mensajes, Métodos, Clases, Variables asociadas y Herencias.
Las clases son plantillas para los objetos. Las clases son como tipos en lenguajes tradicionales de programación. La diferencia es que los usuarios pueden definir nuevos tipos en un lenguaje de programación orientada a objetos. Los tipos más familiares para los programadores incluyen a los enteros, caracteres, números de punto flotante y cadenas (en Pascal). Si una clase es como un tipo en lenguaje de programación tradicional, entonces un objeto es como una variable. Algunas veces se hace referencia a los objetos como instancias de una clase.
Una definición de clase por si misma no crea objetos, si embargo, la definición de clase puede ser usada como un tipo para crear un objeto.

```
Punto p;
```

Este código crea un objeto p, el cual es un punto que tiene valores no definidos para sus elementos internos de datos x,y. Las clases algunas veces son llamadas tipos definidos por el usuario debido a que los tipos definidos por clases se comportan como los tipos integrados de un lenguaje de programación. Nótese que la sintaxis usada para crear un objeto punto no es diferente de la sintaxis empleada para crear un objeto entero con el tipo integrado en C.

int I;

Este código crea un solo objeto ,i, el cual es un entero que tiene un valor indefinido. El proceso de programar en un lenguaje orientado a objetos es el siguiente:

1. Creación de clases que definen la representación de los objetos y su comportamiento.
2. Creación de objetos de acuerdo a la definición de las clases.
3. Arreglos de comunicación entre los objetos como una secuencia de mensajes.

Lo importante de entender es la diferencia entre este tipo de clases y los objetos, los cuales representan instancias de la clase. Las instancias son creadas de la plantilla de la clase y contienen copias privadas de las variables definidas en las clases. Estas variables son frecuentemente llamadas variables de instancia, porque ellas existen en la instancia y no en la clase misma.
Sin embargo los procedimientos asociados con la clase en realidad no existen dentro de las instancias de los objetos. Podría no tener sentido duplicar esto una y otra vez ya que son los mismos para todos los objetos de una clase dada. Los diagramas frecuentemente listarán estos procedimientos dentro del objeto mismo, pero técnicamente ellos residen en otra parte. No obstante, el acceso a estos procedimientos siempre es mediante los objetos creados de la clase en el cual los procedimientos están definidos.

MENSAJES

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a mensajes de otros objetos. Esto marca una clara diferencia con respecto a los elementos de datos pasivos de los sistemas tradicionales. Por ejemplo, cuando un usuario llama a un objeto denominado docu para que escriba un documento. Docu puede enviar un mensaje a otro objeto denominado impre, para que le asigne un lugar en la cola de trabajos de impresión. A su vez, siempre puede enviar un mensaje a docu para requerirle información que posibilite dar formato al documento, y así sucesivamente. Un mensaje puede incluir información para clarificar una petición. Por ejemplo, el mensaje enviado a docu puede incluir el nombre de la impresora. Finalmente, cuando un objeto recibe un mensaje debe conocer perfectamente lo que tiene que hacer., y cuando un objeto envía un mensaje, no necesita conocer como se desarrolla, sino simplemente que se esta desarrollando.
Los objetos se comunican entre sí al enviar mensajes. El termino mensajes se ha empleado de manera no muy formal hasta este punto. En realidad, muchos de los términos de la programación orientada a objetos representan solo ligeros refinamientos de su empleo en el lenguaje diario. Esta fue una elección consciente por parte de quienes diseñaron uno de los primeros lenguajes

orientados a objetos. Se pretendía que las metáforas empleadas para expresar conceptos básicos de programación debían provenir de aquella experiencia al alcance de los niños. Sin embargo, al mismo tiempo las metáforas no podrían restringir el poder expresivo del lenguaje necesario por parte de usuarios adultos avanzados.

Una de las metáforas básicas que se desarrollo fue la noción de que los objetos se comunican al enviar mensajes. Los objetos envían y reciben información al pasar mensajes de una manera semejante a la que emplean las personas. Por ejemplo el torniquete en un restaurante, que se utiliza para pasar información sobre una orden de un mesero a un cocinero. Los mensajes tienen una contraparte denominada método. Mensajes y métodos son los dos lados de una moneda. Los métodos son los procedimientos que se invocan cuando un objeto recibe un mensaje.

En la terminología tradicional de programación, un mensaje es una llamada de un procedimiento a otro. Un método es el código o definición de procedimiento que se invoca. En primera instancia este cambio en la terminología parece arbitrario, incluso trivial. Pero esto se debe a que los lenguajes tradicionales no dan apoyo al polimorfismo y despacho de mensajes que manejan los objetos mismos durante la ejecución. Estos temas se consideraran en el momento adecuado. Mientras tanto, la terminología de transferencia de mensajes va a hablar acerca de la construcción de sistemas que directamente son un modelo o simulan contrapartes del mundo real.

Los métodos y mensajes ayudan a hacer obligatoria la división del trabajo. El sistema de transferencia de mensajes por medio de un torniquete en el restaurante traza una frontera y define la responsabilidad funcional. La frontera conserva una división clara entre el trabajo que realiza el mesero o mesera y el que efectúa el cocinero. El mesero envía un mensaje al cocinero por medio del torniquete. El objeto receptor efectúa un método en respuesta a un mensaje. Un método es como una receta . El cocinero dispone de un cierto numero de métodos para preparar diferentes ordenes. El cocinero sigue un método para preparar una orden de huevos revueltos al recibir el mensaje adecuado. La distinción entre métodos y mensajes conserva al mesero en el comedor y al cocinero en la cocina.

Un ejemplo es una simulación de actividades o tareas que tienen lugar en un restaurante común. Los objetos básicos en la simulación son mesas, clientes, meseros, cocineros, el torniquete y la barra . Lo mas importante es que todos los objetos en la simulación están conectados a través de mensajes. Los mensajes proporcionan a los objetos comunicación a través de estrechas interfaces de ancho de banda. El pasar mensajes reduce el número de conexiones entre los componentes del sistema. Reducir el número de conexiones incrementa la modularidad de los componentes del sistema.

Otra cosa que quizá no resulte obvia de inmediato es que los objetos pueden contener otros objetos . Por ejemplo, una mesa contiene un conjunto de cubiertos y un torniquete contiene un conjunto de papeletas, tickets, con ordenes. La forma en que un objeto se comunica con otro depende de la interfaz. Por ejemplo, un mesero puede comunicarse en forma directa con cada cliente que se encuentre

sentado a una mesa. En la simulación, la mesa indicaría al mesero cuantos clientes se encuentran ahí al devolverle una lista de clientes. Entonces, el mesero tomaría la orden de cada miembro de la lista.
Sin embargo, la interfaz con el torniquete es un poco distinta. Lo único que puede hacer un mesero con un torniquete es agregar un ticket de orden. Lo único que puede hacer un cocinero con el torniquete es quitar el siguiente ticket. Este torniquete funciona mas como una pila que como un verdadero torniquete debido a que no puede efectuarse el acceso directo a los elementos individuales del torniquete (tickets)..
Hasta aquí nos damos cuenta que los objetos envían y reciben mensajes. También sabemos que los mensajes son llamadas de procedimiento. Además, los mensajes llevan información. Lo hacen a través de argumentos y valores resultantes, igual que lo hacen los procedimientos. Por ejemplo, un mesero podría enviar un mensaje que llevará información del tipo de un argumento al torniquete:
En esta expresión, un ticket es un argumento al mensaje agregar ticket, el cual se envía al objeto torniquete. Los mensajes también pueden devolver información. Por ejemplo, un mesero podría ver en la barra para saber sí una orden se encuentra preparada. Esta actividad se simularía al hacer que el mesero preguntara a la barra si hay una orden lista.

```
if (LaBarra.HayPedidoListo() )
 unPedido = LaBarra.RetirarPedido();
```

Los dos mensajes HayPedidoListo y RetirarPedido devuelven información al mesero. El primero da como resultado un valor booleano que indica sí la orden esta lista. El segundo da como resultado un objeto propiamente dicho, el cual es la orden. Ahora, el mesero puede llevar a cabo otro método para entregar la orden en la mesa adecuada. Además de llevar información, los mensajes también pueden invocar procesamiento.

METODOS

El procesamiento está asociado en forma directa con los métodos. Según se mencionó antes, METODO es tan solo otro nombre para PROCEDIMIENTO. Algunas veces a los métodos se les denomina funciones miembro porque son miembros de un objeto. Hasta ahora, todos los métodos que hemos visto son públicos. Pero los métodos también pueden ser privados. Por lo general, los métodos públicos no están asociados con el concepto de procesamiento. La función usual de los métodos públicos es proporcionar acceso a las variables de instancia de un objeto y recibir mensajes. Solicitar acceso a datos privados es una forma de procesamiento, pero solo en el sentido técnico de la palabra.
Un proceso muy complejo puede invocarse a través de mensajes simples. Por ejemplo, un objeto directorio puede recibir la solicitud de que se actualice a si mismo. Un mensaje de solicitud de este tipo invocaría un procesamiento en el

interior del objeto, mucho mas complejo que la mera recolección de valores de las variables de instancia . En este caso, un método de interfaz pública podría llamar a uno o mas métodos privados que hicieran todo el trabajo sucio.
Un ejemplo de métodos privados es dentro del objeto cocinero. Nadie puede comunicarse directamente con el cocinero. El cocinero revisa de tiempo en tiempo el torniquete en busca de un pedido. Una vez que un pedido está en el torniquete, corresponde al cocinero tomarlo. Después tiene tres distintos métodos privados que efectuar con base en la información que contiene el pedido. Este es exactamente el tipo de procesamiento para el que se diseñaron los objetos.
La interfaz con el objeto cocinero es estrecha. Solo hay dos canales a través de los cuales puede fluir información (u objetos). Una interfaz estrecha impide que las líneas de comunicación se crucen en el sistema complejo. Una vez que un objeto toma el control para procesar un mensaje, es responsabilidad de ese objeto efectuar el método adecuado como respuesta. Los objetos del exterior no pueden saber y no les importa cómo se efectúa este procesamiento.
El procesamiento y su relación con el envío de mensajes entre objetos pueden comprenderse en términos de conceptos de programación tradicional. Los objetos contienen procedimientos. A los procedimientos que se encuentran en el interior de los objetos, se les denomina métodos. Los métodos responden a mensajes y procesan información. Son código ejecutable.
Un método reside en un objeto y determina como tiene que actuar el objeto cuando recibe un mensaje. Adicionalmente, las variables asociadas permitirán almacenar información para dicho objeto. Un método puede también enviar mensajes a otros objetos solicitando una acción o información. La estructura más interna de un objeto esta oculta para otros usuarios y la única conexión que tiene con el exterior son los mensajes. Los datos que están dentro de un objeto, solamente puede ser manipulados por los métodos del propio objeto. Un programa orientado a objetos en ejecución realiza fundamentalmente tres cosas:

1. Crea los objetos necesarios
2. Los mensajes enviados a unos y otros objetos dan lugar a que se procese internamente la información.
3. Finalmente, cuando los objetos no son necesarios, son borrados, liberándose la memoria ocupada por los mismos.

SUBCLASES

Una clase es 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 la clase. Precisamente 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. Un objeto es una variable del tipo definido por una clase. Dicho de otra forma, una clase es un tipo de objetos definido por el usuario.

Un objeto es creado cuando a la clase correspondiente recibe un mensaje solicitando su creación.
Una clase puede también agrupar elementos comunes en conjuntos que denominaremos subclases.
Las subclases también reciben el nombre de clases derivadas. Como es en el caso de una estructura de árbol a la que da lugar un análisis.
Una clase es un tipo definido por el usuario. La definición de una clase especifica como son los objetos de esa clase, esto es de que miembros constan y el conjunto de operaciones que pueden aplicarse a tales objetos.
En C++ la definición de una clase consiste en dos partes. La primera esta formada por el nombre de la clase precedido por la palabra reservada class.
La segunda parte es el cuerpo de la clase encerrado entre llaves y seguido por un punto y coma o por una lista de identificadores de objetos de esa clase, separados por comas.

```
class nombre_clase
        cuerpo de la clase
        lista de objetos
```

1.2 ENCAPSULAMIENTO

La combinación de datos y procedimientos juntos en una declaración de objetos se denomina encapsulamiento o encapsulación. Los lenguajes que aplican programación orientada a objetos poseen diferentes grados de encapsulación. Un objeto y su encapsulación asociada el siguiente ejemplo ejemplo en Pascual muestra:

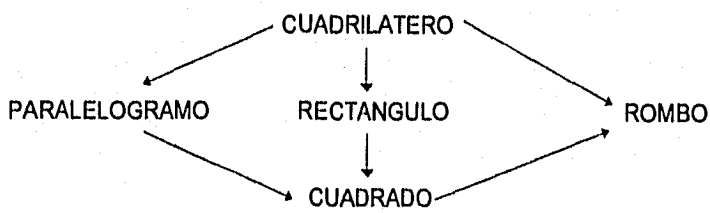
```
TYPE Emplazamiento = OBJECT
        x,y : Integer;
        PROCEDURE Posicionar (Ptx, Pty : Integer);
        FUNCTION LeerX  : Integer;
        FUNCTION LeerY  : Integer;
           END;
```

Es la manera de ocultar los detalles de la representación interna de un objeto presentando solamente la interfaz disponible al usuario. Este mecanismo es indispensable si se piensa en la creación de componentes de software reusable.
Aunque, como se ha comentado, usted puede acceder a cualquier dato de un objeto, siempre que lo desee, las reglas de POO recomiendan que se debe acceder a los campos de datos solo a través de procedimientos y funciones. En otras palabras se deben desarrollar procedimientos y funciones adecuadas para todas las operaciones de inicialización, modificación, entrada, salida y de comparación que deben realizarse con los datos del objeto. En esencia, un objeto son datos privados a los que se accede a través de subprogramas públicos.

Uno de los objetivos esenciales de la POO es como ya se ha comentado el encapsulamiento o encapsulación : agrupar datos y el código o métodos que los manipula en una entidad llamada objeto, es decir, crear objetos que funcionen como unidades completas. Una de las reglas del encapsulamiento es que el programador nunca necesita acceder directamente a los campos de datos del interior de un objeto.
Los objetos proporcionan un nivel de abstracción próximo a la realidad, que permite manipularlos sin conocimiento de lo que se encuentra en su interior (es decir, el código de manipulación de los datos). Se puede comparar esta noción de encapsulamiento con lo que le sucede a un conductor de automóviles. Puede avanzar, girar, retroceder, detenerse, etc. sin necesidad de conocer la mecánica del automóvil. La interfaz del conductor con el vehículo esta constituido por ordenes disponibles en el habitáculo del automóvil (volante, pedales, palanca, etc).
Uno de los objetivos primordiales de la POO es la encapsulación, es decir, la creación de objetos que funcionan como entidades completas. Una de las reglas de la encapsulación es que el programador nunca necesita acceder directamente a los campos de datos de un objeto. En vez de esto, se deben definir métodos dentro del objeto que gobiernen toda la manipulación de datos.

1.3 HERENCIA

La herencia es una de las propiedades mas notables de la programación orientada a objetos. Cuando un objeto se deriva de otro objeto, se obtienen todas las propiedades de este otro.
En otras palabras, un objeto puede heredar propiedades de otro objeto. El objeto padre se denomina ascendiente y el objeto hijo, descendiente. Un tipo objeto puede tener varios tipos descendientes directos, estos a su vez también pueden tener varios tipos descendientes directos, estos a su ves pueden también tener descendientes que heredan del mismo tipo original. Un nuevo objeto se puede basar en otro objeto definido anteriormente, en cuyo caso hereda todas las propiedades de los campos del viejo objeto. Los objetos tienen una gran aplicación en el mundo de los gráficos. La siguiente figura representa un objeto denominado cuadrilátero que tiene una estructura jerárquica con una serie de tipos descendientes del mismo.



Tipo Objeto cuadrilátero. Un tipo objeto no puede tener por ahora, más que un ascendiente directo, en lo que se llama herencia simple. Así por ejemplo, si triángulo es un objeto, se pueden definir diferentes nuevos objetos.
La herencia es transitiva: Un objeto hereda propiedades de su ascendiente directo y después de los ascendientes de nivel superior. Cuando varios tipos objetos están enlazados entre ellos por herencia, están organizados en lo que en teoría de Grafos de denomina un Grafo orientado (la relación de herencia tiene un sentido: un tipo objeto es el padre y otro es el hijo).Cuando la herencia es múltiple, este grafo es un árbol. En la fase de concepción de una aplicación que comporta objetos, se deben determinar los tipos objetos a definir y sus relaciones de herencia. Una etapa consistirá en clasificar los objetos de modo jerárquico. La herencia simple permite que los objetos tengan muchos descendientes, pero sólo un ascendiente inmediato o directo. Es el mecanismo para compartir automáticamente métodos y datos entre clases, subclases y objetos. Aunque los objetos contienen sus propios métodos y datos, también pueden heredarlos de otros objetos. La herencia en la POO tiene como raíces un concepto que es familiar a los programadores de Turbo Pascual: los registros anidados.

HERENCIA SIMPLE Y MULTIPLE

A) Herencia Simple. Permite crear (derivar) una nueva clase basándose en otra clase más general. Una clase derivada adquiere todas las propiedades y métodos de la clase de la que se derivó (clase base), pudiendo añadir nuevos elementos o modificar ligeramente los ya existentes.

B) Herencia Múltiple. Proporciona la posibilidad de derivar una clase de un conjunto de ellas.
La diferencia entre la composición y la herencia múltiple consiste en que en la primera, se realiza una fusión de las clases base generando una clase autosuficiente. En la segunda se mantiene una liga con sus clases base, de tal forma que si utiliza algún método y no lo tiene, lo solicita a las clases de las cuales se derivó.
Las clases no existen en un vacío. Cuando en la definición de una clase **B** en la forma

```
class B
   ( //.....
     A x;
     //....
```

se usa una componente de otra clase **A,** o tiene una función que utiliza a la clase **A,** se dice que la clase **B** *es usuario de la clase* **A**. El encapsulamiento no permite que **B** haga uso de las partes privadas de un objeto de la clase **A**. Esto es correcto desde el punto de vista de esta relación de *clientilismo,* pues nos garantiza que **B** pueda usar la clase **A** pero respetando una funcionalidad ya

existente, aumentando con ello la confiabilidad . Sin embargo, no es suficiente desde el punto de vista de la *extensibilidad. Es decir , también es conveniente poder adaptar o modificar, a nuevos requerimientos , clases ya existentes.*
Vamos ahora a definir clases por combinación , extensión y especialización de otras clases pero sin dar al traste a lo logrado con el encapsulamiento . Esto se ha logrado con el encapsulamiento. Esto se logra en la **POO** a través del concepto de **herencia**. Mediante el recurso de herencia se puede definir una clase como **heredera, descendiente o derivada de otra(s) clase(s)** que se denominará clase **padre** clase **base.**
El término herencia ha sido adoptado por analogía con las ciencias naturales, donde la clasificación, basada en un árbol de jerarquías de herencia, ha demostrado ser exitosa. Al definir que una clase *hereda o deriva* de una o varias clases estamos estableciendo una jerarquía de descendencia en la que las clases más generales, más arriba en la jerarquía, expresan una funcionalidad común a todas sus *clases descendientes* que a su vez *añadirán* (extensión por ampliación) nuevas funcionalidades y/o *adaptarán* (extensión por especialización) algunas de las existentes.
La herencia aporta una interpretación significativa a ambas perspectivas. Si una clase hereda de otras , extiende y/o mejora los recursos que ésta ofrece como módulo. Desde el punto de vista de tipo la herencia introduce una relación de subtipo is- a entre la clase heredera o derivada y la clase base., es decir un objeto de la clase heredera puede verse como un objeto de la clase base.

HERENCIA SELECTIVA

Por regla general a través de la herencia "se transmiten todos los recursos y propiedades".
En ocasiones se puede querer heredar de una clase pero "no heredar" algunos recursos de la clase base. Es decir heredar de una clase pero no permite a los usuarios de la nueva clase el acceso a alguno de los recursos.
El concepto de herencia no podría ser utilizado en toda su potencialidad si tuviéramos que heredar de una clase. Heredar de una sola clase impondría una estricta jerarquía de descendencia expresada en forma de árbol. En ciencias más "artificiales ", como el software, esta clasificación es insuficiente.
El problema es que un objeto puede clasificares desde varios puntos de vista, es decir desde más de una forma.
Una jerarquía de herencia simple nos limitaría en el enfoque descendiente (top bottom) que precisamente promueve la POO. ¿Cómo podríamos aprovechar la existencia de varias clases, y de las cuales queremos tomar distintos recursos de cada una, en la formación de una nueva clase? . No puede pensarse en mirar atrás, hacer un nuevo diseño y cambiar las clases ya existentes porque esto es contrario al principio de extensibilidad y reusabilidad que se quiere promover.
Varias versiones de SMALLTALK, considerando uno de los lenguajes precursores en la POO , no incluyen herencia múltiple. Algunos autores argumentan esta decisión basados en los conflictos que la herencia múltiple puede introducir.

Se plantea que el efecto de la herencia múltiple puede lograrse con herencia simple. Es decir por ejemplo si tenemos dos clases A y B y quisiéramos definir una clase C que heredase de ambas podríamos definir C en la forma:

```
class C: public A
      public:
        B x;
      //...
```

es decir, definir una componente en C que sea de tipo B y heredar sólo de A. De esta manera cuando se tenga una variable c de tipo C podrían accesarse los recursos de B mediante la componente x de c en la forma

c.x.F(...) donde F es el recurso en B

Sin embargo, esta notación, además de más engorrosa, no es natural porque impone al cliente de C el conocimiento de la componente x..
La sintaxis para expresar herencia múltiple es una generalización de la herencia simple. En herencia simple cuando tenemos que una clase C hereda de una clase A denotamos esto en la forma

```
class A
    //...

class C: public A
    //...
```

Con herencia múltiple si tenemos las clases

```
class  A1
    //...

class  A2
    //...

class  A3
    //...
```

y queremos definir una clase C que herede de A1, de A2, y de A3, debe definirse C en la forma

```
class: public A1, public A2, public A3
   //...
```

Al igual que en herencia simple puede no usarse la palabra reservada **public** para especificar el acceso a alguna de las clases de las cuales se hereda, en este caso se interpreta por omisión como **private.** En la lista de las clases bases (de las que se hereda) no puede especificarse un mismo nombre de la clase más de una vez.

UNION DE CLASES

Otro uso de la herencia múltiple es para definir clases por unión o combinación de varias clases. De esta manera un objeto de la clase derivada de la misma puede verse desde distintos puntos de vista.

HERENCIA, POLIMORFISMO

Además de la encapsulación, la programación orientada a objetos hace uso de la herencia y el polimorfismo. La herencia involucra la clasificación de objetos de acuerdo a propiedades compartidas. El polimorfismo involucra el envío de mensajes a objetos de tipo desconocido.
La programación orientada a objetos envía mensajes a objetos de tipo no conocido con exactitud. No obstante que el programador envía estos mensajes polimorficos, sabe que todos los objetos en una colección polimórfica tienen algo en común. Por lo menos, los objetos polimorficos comparten un conjunto de nombres en su interface pública, a este conjunto común de nombres se le denomina como el protocolo de comunicación de los objetos.
Los objetos pueden ser clasificados de acuerdo con los rasgos que ellos comparten. Aunque un conjunto de objetos podría definir un conjunto común de nombres de métodos para un protocolo de comunicación, algunos objetos pueden proporcionar métodos que otros objetos no proporcionarán. Más aún, aunque los nombres de los métodos sean los mismos, las definiciones de los métodos podrían variar. Por ejemplo, un método de dibujo para un circulo podría tener que ser implementado diferentemente que un método de dibujo para un rectángulo.
El polimorfismo explota las características que un conjunto de objetos comparte. Por lo tanto, si se tiene una colección consistente solo de círculos y rectángulos, se puede enviar con seguridad mensajes de dibujo a cualquier elemento de la colección. La respuesta, sin embargo, será ligeramente diferente dependiendo si el receptor del mensaje es un círculo o un rectángulo. La herencia proporciona la herramienta para expresar las propiedades del protocolo de comunicación comunes a un conjunto de objetos. El polimorfismo asegura que los objetos respondan a los mensajes generales en sus caminos particulares.

COLECCIONES POLIMORFICAS

Considerando la forma en que el polimorfismo se aplica a una colección de ventanas en una pantalla de computadora. Las ventanas tienen una amplia variedad de aspectos y formas. Esto quiere decir que hay muchos tipos distintos

de ventanas. Una de las primeras cosas que hace cuando ve un conjunto de cosas nuevas es ajustarlo en un marco que tenga sentido para una persona. Luego, intenta encontrar la forma para clasificar nueva información de manera que esto le ayude a asignarle un significado. Una manera de hacerlo consiste en buscar similitudes entre los objetos.

Un ejemplo de la propiedad común de las ventanas es el menú de control. Todas las ventanas de nivel superior contienen menús y responden a los mensajes que se envían al efectuar una selección de menú. Al dar un clic al botón izquierdo del ratón en cualquiera de los iconos provocaría que apareciera el mismo menú. El menú asociado con un icono especifico siempre aparecerá directamente encima de este. Así, los iconos se comportan diferente al especificar otras ubicaciones para presentar sus menús. Aunque se trata de diferencia menor en el comportamiento, ésta pequeñez resulta importante si el usuario desea comprender un sistema de ventanas, y critica si esta tratando de programar uno.

La Herencia y el Poliformismo son herramientas que organizan los tipos de jerarquías y simplifican la comunicación. La herencia permite que la definición formal de lo común se exprese para un conjunto de objetos, mientras que el polimorfismo permite usar un protocolo común cuando se comunica con objetos del mismo tipo de jerarquía.

Como se estableció anteriormente, una clase de objetos es un conjunto de objetos que comparten rasgos y características comunes. Estos rasgos y características están definidos en base a la clase. Los subsecuentes tipos y clases están definidos como especializaciones de la clase base. Como programador se puede crear nuevos tipos o clases que hereden sus características de la clase base. Estas clases nuevas son clases derivadas.

El proceso de definición de nuevos tipos y rehuso de códigos previamente desarrollados en base a las definiciones de la clase es llamado programación por herencia. Las clases que heredan propiedades de una clase base pueden en turno servir como definiciones de base para otras clases. Las jerarquías de tipos normalmente se establecen en estructura de árbol, con las definiciones de propiedades generales de las clases en la raíz y las propiedades especificas en las hojas. Este árbol se conoce como jerarquía de clases o jerarquía de tipos.

Considere el desarrollo de un tipo de jerarquía para un conjunto de formas. El objetivo final es desarrollar la capacidad de rotar un conjunto de formas sobre un display alrededor de un punto de referencia. El punto de referencia será el origen Punto (0,0). El código del programa para este ejemplo se presenta en C++. Para hacer portátil el código de la jerarquía de la clase Shape a través de los diferentes compiladores y archivos gráficos C++ se utilizará una pantalla abstracta. Solo un ejemplo de la clase de pantalla se usa en el programa dado. El nombre global dado a éste ejemplo global de la clase Screen es pantalla. La clase pantalla define una pantalla abstracta cuyas coordenadas x, y varían del Punto (0,0) al Punto (100,100). El Punto (0,0) se encuentra en la esquina inferior izquierda de la pantalla y el Punto (100,100) en la parte superior derecha de la pantalla.

La clase pantalla provee métodos para limpiar la pantalla, establecer los colores interno y externo así como las líneas de dibujo. La clase pantalla es un buen

ejemplo de como la encapsulación de la función y los datos pueden maximizar la libertad del programador -en este caso el cambio de la representación interna de una pantalla real.
La jerarquía de la clase Formas desarrollada aquí presenta un ejemplo simplificado de compromiso entre los objetos de la pantalla que tienen propiedades comunes.
Los objetos de ejemplo de la jerarquía simplificada de la clase SHAPE son círculos, rectángulos y triángulos. Habiendo identificado los objetos que serán encontrados en la aplicación final, se puede seguir al siguiente paso: buscar las propiedades comunes de círculos, rectángulos y triángulos en el contexto del dibujo de estas formas en la pantalla. Eventualmente ellos rotaran alrededor del Punto (0,0) un ángulo especificado.

COMPROMISOS ENTRE CLASES Y CLIENTE

El programador que trabaja en el paradigma de objetos (que se basa en instrumentar clases, crear objetos a partir de las clases y enviar mensajes a los objetos) puede situarse en dos puntos de vista con respecto a una clase:

- Como cliente o heredero de la clase. En este caso humildemente y de buen corazón confía en que la clase que va a utilizar, o de la cuál va a heredar, esté bien interesada. Es decir, supone que esta no le generará errores si el cumple los requisitos que la clase exige para su uso.
- Como instrumentador de la clase. En este caso autosuficientemente considera que el desarrollará una clase correcta, que no le traerá errores a sus usuarios si éstos cumplen disciplinadamente con los requisitos de utilización de la misma.

En el desarrollo de un aplicación estos puntos de vista establecen un proceso en espiral que tiene dos extremos.

- Cuándo se utilizan clases y funciones ofrecidas por el propio constructor del compilador (incluso algunas de ellas inmensas dentro del propio compilador como las tipos simples y sus operaciones). En este caso la confianza debe ser máxima (aunque no absoluta, porque ningún compilador está exento de fallas).
- Cuándo se utiliza el objeto raíz cuya creación desencadenará la ejecución de la aplicación.

El éxito de un sistema se basa en que cada parte cumpla su compromiso en este proceso. El instrumentador de la clase en lograr que la clase cumpla lo que promete y el cliente de la clase en usar la clase en la forma en que le piden.
Como somos humanos cada parte puede incumplir con su trato. Los lenguajes con fuerte control estático de tipos (static strong type checking), como es en caso de el lenguaje C++, tratan de detectar la mayor parte de estos incumplimientos en tiempo de compilación, basados en una utilización inconsistente del sistema de tipos.

ASERCIONES

Hay otros compromisos de ambas partes. Por ejemplo, el cliente de stack no deberá hacer s.Push(x) si la pila s está llena. El programador de stack debe garantizar que si aplica la función Top a s, luego de haber hecho s.Push(x), ésta debe retornar x. Este tipo de compromiso no es controlable estáticamente a partir del sistema de tipos. La teoría de verificación y especificación de programas introduce el compromiso de las aserciones para expresar éstos.
Podemos pensar que en el programa la clase tiene cierta ventaja sobre el que la usa pues podrá programar el control de que el cliente cumpla su parte, mientras el cliente sólo puede tener fe en que el programador de la clase le reportará su incapacidad de cumplir con sus promesas. Este desbalance se compensa porque, como vimos anteriormente, en el proceso de desarrollo de un proceso completo la posición programador_cliente o programador _vendedor es relativa: quién es cliente de unos es vendedor de otros.
En mundo ideal, como todas las partes tienen buena voluntad y buena capacidad no debieran ocurrir estas violaciones. En la práctica real ellas pueden ocurrir excepcionalmente (de ahí el término excepción) porque:

- Conscientemente una de las partes no controla que cumple lo que le exige la otra. Probablemente lo haga para no "llenar" el código de controles (no vamos a estar verificando si el divisor es cero antes de manda a hacer cada división).
- Pueden ocurrir fallas, que no se podían controlar anticipadamente, en componentes a las que no se les puede reclamar: hardware, sistema operativo, biblioteca básica del compilador.

En definitiva somos humanos y cometemos errores de programación.

1.4 POLIMORFISMO

Permite la manipulación de objetos de clases distintas como si fueran de la misma clase, con lo cual es posible definir interfaces uniformes para diferentes tipos de objetos.
Las entidades internas de un programa deben poder manejar conjuntos de objetos de diferentes clases de la misma forma en que manejan conjuntos de objetos iguales.
Una operación puede comportarse de varias formas de acuerdo a la clase de objetos que se manipulan.
De igual forma, un objeto puede comportarse de diversas formas dependiendo de los tipos de parámetros que reciba.
Los objetos que se presentan en una pantalla podrían ser barras de desplazamiento, ventanas, iconos, menús, botones u otros controladores, que son conocidos como objetos polimorficos. Esto significa que ellos comparten ciertas características como la habilidad de ser puestos en una colección sencilla (una pantalla por ejemplo) y la habilidad de responder a los mismos mensajes.

El polimorfismo esta relacionado directamente con la herencia. La programación por herencia permite definir nuevos objetos al describir la forma en que difieren de objetos ya existentes. Por ejemplo, en Smalltalk ,la ventana de petición de entrada se comporta como la ventana del editor de texto , excepto que la ventana de petición de entrada solo retiene una única línea del texto.
Cuándo se han leído algunos libros de programación orientada a objetos se puede observar la existencia de los términos encapsulación, herencia y polimorfismo. Por lo tanto existe una relación omnipresente del triángulo encapsulación herencia polimorfismo ilustrado o proyectado en la pantalla. La herencia y el polimorfismo son mas complejos y requieren el entendimiento de algunos conceptos preliminares.
La encapsulación, por otro lado se puede discutir dentro de los conceptos preliminares.
Aunque se debe empezar con objetos, la finalidad que se desea es el marco o la estructura de programa. En lugar de considerarse como un conjunto de procedimientos, un programa orientado a objetos se comprende mejor como un conjunto de objetos en comunicación. Destacar los objetos, en vez de las acciones, lleva a una estructura de programa muy diferente de la representativa en los programas orientados a procedimientos.
Las buenas técnicas de estructuración para la programación procedural están ya establecidas. Los procedimientos son los bloques constructivos básicos en programación procedural. Una técnica de uso común es el refinamiento descendente por pasos (top-down stepwise refinement), en el cual se empieza con la visión mas general posible de la estructura de un programa y descompone este programa en piezas manejables más pequeñas: los procedimientos. Esto es, empieza por construir lo que desea que el programa haga en una sola proposición: "Necesito un programa que cuente el número de palabras de un archivo." Posteriormente se va a expresar una solución abstracta preliminar al describir los datos generales necesarios para resolver el problema, por ejemplo:

1. Abrir el archivo.
2. Leer la siguiente parte del archivo.
3. Incrementar un contador de palabras.
4. Al final del archivo informar el número de palabras leídas.
5. Cerrar el archivo.

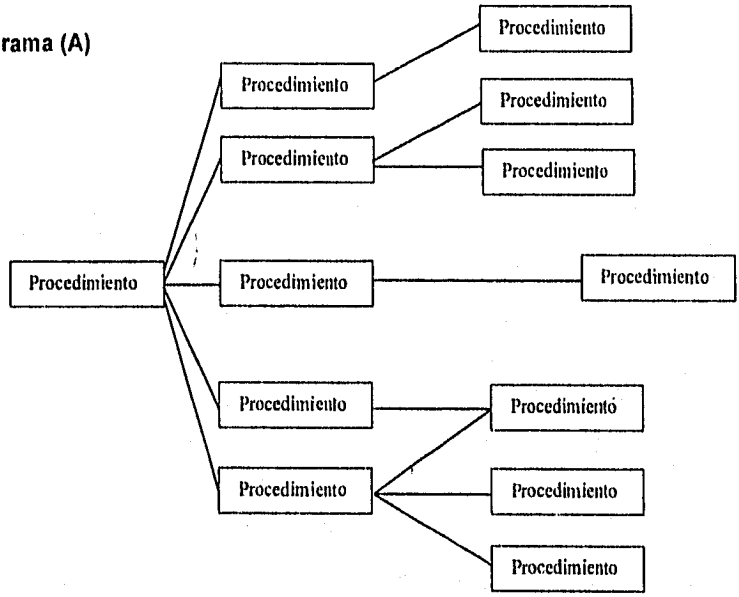
La estrategia consiste en continuar descomponiendo cada subtarea hasta que, con el tiempo, se llegue al punto en que se pueda expresar los pasos para la tarea dada con proposiciones del lenguaje de programación deseado. La mayoría de los programas procedurales es mas complicada que un programa para contar palabras, pero el proceso para descomponer un problema complejo en subtareas mas manejables es el mismo, sin importar la complejidad. Empezando a partir de abstracciones generales, se puede descomponer el problema poco a poco en pasos que se vuelven cada vez más concretos. A partir de este análisis se desarrollan niveles de abstracción. En la solución final de programación, se tendrá

un procedimiento más general que llama a procedimientos mas concretos también llamadas subrutinas hasta que por último llegue al nivel más bajo de abstracción, el cual puede expresarse directamente como proposiciones del lenguaje de programación.

Esta estructura es ideal para la programación procedural. Los datos locales se encuentran ocultos en el interior de procedimientos. Los datos compartidos se pasan como argumentos.

Esta descomposición jerárquica de un problema va a erigir una estructura semejante a la que aparece a continuación :

Diagrama (A)



Las técnicas de descomposición jerárquica y de refinamiento por pasos son tan comunes que la mayoría de los programadores las utilizan sin esfuerzo consciente: en forma automática buscan los procedimientos en un problema dado. Pero las dos nociones que subyacen en esta técnica son cuestionadas por el enfoque orientado a objetos. La primera noción es que la programación debe

enfocarse de arriba hacia abajo, y la segunda es que los programas sean una serie de procedimientos o acciones que se invocan uno después de otro. Podemos hacer una comparación con la representación del programa procedural que aparece en la **Diagrama (A)** con el **Diagrama (B)** que representa un programa orientado a objetos.
Este tipo de estructura es ideal de programa para la programación orientada a objetos. En donde un programa es un conjunto de objetos que se comunican por medio de mensajes.
Es fácil ver la diferencia de estructura. Pero la segunda figura también señala varias otras cosas acerca de la programación orientada a objetos.
En primer lugar, se muestra que los objetos son las unidades estructurales primarias del programa. Esto por si mismo no indica mucho, ya que la estructura exacta de los objetos todavía no se ha abarcado. Sin embargo, la tendencia es a describir los objetos por sustantivos o nombres en lugar de emplear verbos. Los objetos en el espacio del problema se asocian o mapean mas directamente con objetos en el programa que con procedimientos.

Los programas orientados a objetos con frecuencia se describen como simulaciones debido a que los objetos en el programa por lo general imitan el comportamiento de sus contrapartes del mundo real.

Una segunda observación, que resulta menos obvia a partir del **diagrama (B),** es que también se puede considerar a los objetos como tareas o procesos. Entre los ejemplos de objetos de este segundo diagrama se encuentran "Teclado", "Ratón", "Despachador de Control" y "DiskBrowser".

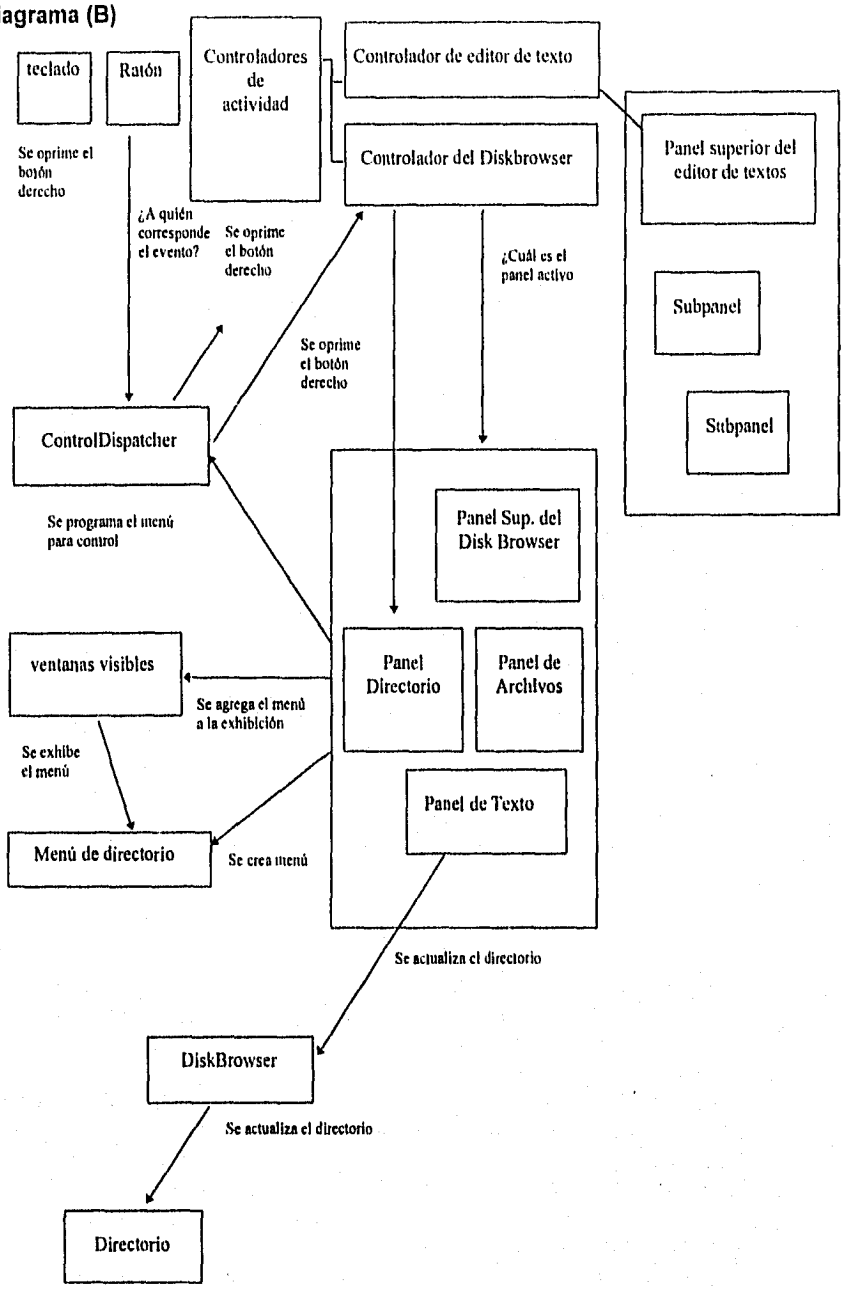
El procesamiento tiene lugar en el interior de los objetos y la información se pasa en uno u otro sentido entre objetos.

Los objetos pueden invocar procesamiento en otros objetos al enviar mensajes.
Son ejemplos de mensajes "Botón derecho oprimido", "Actualizar directorio", "Crear menú" y "Exhibir menú". Lo más importante, es que un programa orientado a objetos es un conjunto de tareas que se comunican.

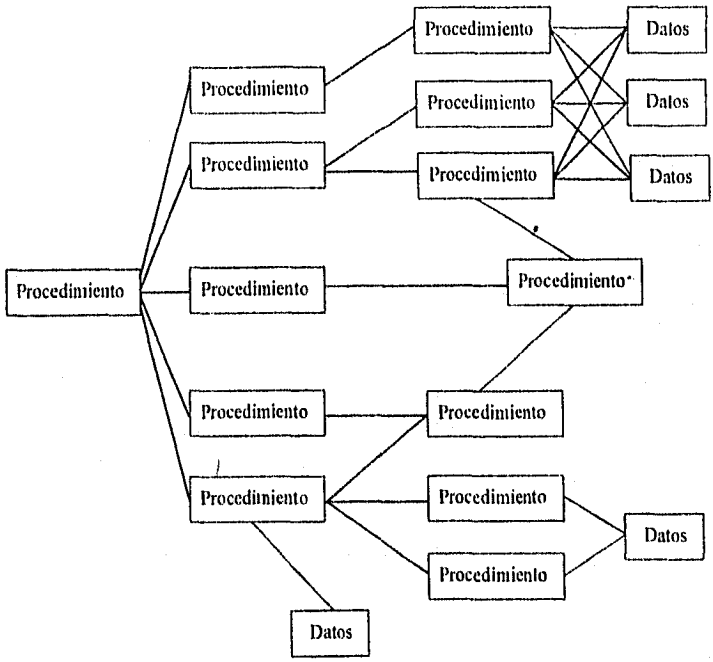
El programa cuyo diagrama aparece en la segunda figura es un Diskbrowser Smalltalk . En realidad, éste es solo un diagrama parcial del programa.
Un DiskBrowser consiste en cinco ventanas, de las cuales solo una se representa en el diagrama.

El diagrama muestra la comunicación que tendría lugar entre algunos de los objetos del escritorio Smalltalk si se oprimiera el botón derecho del ratón mientras se apunta al panel de la lista de directorio de un objeto Diskbrowser.

Diagrama (B)


teclado
Ratón
Controladores de actividad
Controlador de editor de texto
Controlador del Diskbrowser
Panel superior del editor de textos
Subpanel
Subpanel
Se oprime el botón derecho
¿A quién corresponde el evento?
Se oprime el botón derecho
¿Cuál es el panel activo
Se oprime el botón derecho
ControlDispatcher
Se programa el menú para control
Panel Sup. del Disk Browser
ventanas visibles
Panel Directorio
Panel de Archivos
Se agrega el menú a la exhibición
Se exhibe el menú
Panel de Texto
Menú de directorio
Se crea menú
Se actualiza el directorio
DiskBrowser
Se actualiza el directorio
Directorio

En la siguiente figura vamos a observar que mientras en la primera figura se representa lo ideal, en esta ultima figura se representa la práctica real de la programación.



Lo ideal aboga por ocultar información lo más posible para mantener modulares a los componentes de un programa. Los datos locales se ocultan dentro de procedimientos y la información que necesitan compartir los procedimientos se pasa en forma de argumentos procedurales.
Sin embargo, algunas veces el número de procedimientos que necesitan compartir información es demasiado grande o las estructuras mismas de datos son demasiado grandes para poderse pasar con facilidad de un procedimiento a otro. En tales casos, se emplean datos globales. El problema al utilizar datos globales es que cualquier procedimiento contenido en el programa se puede alterar. Esto facilita que surjan errores ocultos. Y dificulta detectar esos errores porque el alcance de las proposiciones que podrían provocar un problema

observado es ahora todo el programa. Los conceptos de ocultamiento de datos y de abstracción de datos se representaron para resolver el problema del acceso irrestricto a los datos globales. La programación orientada a objetos reconoce la tendencia a usar datos globales y proporciona herramientas para hacer explícitas las relaciones entre datos globales y procedimientos.
Un método estático es aquel que esta determinado completamente cuando el programa se compila. Una cabecera de un método estático en una definición de objeto es idéntico al formato de la cabecera de procediendo o función en una sección de interface de una unidad.
Las llamadas a los métodos estáticos se enlazan a los métodos que llaman en tiempo de compilación. Este proceso se denomina ligadura estática o temprana. Con los métodos virtuales, la conexión no se realiza hasta que el programa se esta ejecutando y se envía realmente la llamada., se dice entonces que existe ligadura dinámica o tardía. Ligadura dinámica significa simplemente que el camino acto de ejecución se determina en tiempo de ejecución y no en tiempo de compilación.
La ligadura dinámica permite definir instancias o variables de objetos polimorficos es decir, cuyo tipo será definido en la ejecución entre los disponibles en una jerarquía de objetos., es decir, una variable objeto de sus descendientes.
El polimorfismo permite decidir a un programa en tiempo de ejecución a que método se llama.

POLIMORFISMO Y HERENCIA SELECTIVA

La introducción de este recurso de herencia selectiva puede traer inconsistencias en el sistema de tipos cuando se combina con el polimorfismo. Supongamos que un cliente de la clase Ave tiene una función

```
void Prueba_de_Vuelo(Ave &x)
 //...
 if Condition x. Vuela( );
Qué pasaría si se tiene
   Pingüino p;
   //...
   Prueba_de_Vuelo (p);
```

Una vez más el programador optimista pediría que le dejen hacer la prueba de vuelo pues es posible que la condición no se cumpla. Sin embargo, este optimismo puede provocar la muerte del pobre pingüino.
Este hecho de que la misma notación tenga distintas formas (interpretaciones) es denominado *polimorfismo* (del latín varias formas).
De esta forma el concepto de herencia introduce una nueva relación de subtipo entre las clases: la relación Is-a (es-un). Todo objeto de una clase derivada puede verse (es un) como un objeto de la clase base que puede llevar a cabo todas las

mismas acciones (posiblemente mejoradas) que el objeto de la clase base puede realizar.
La habilidad de poner objetos de una base común en un arreglo o colección simple y el uso de un protocolo común definido en la clase base para comunicación con los objetos individuales, es lo que se conoce como polimorfismo. El termino polimorfismo viene de Biologia y significa literalmente, muchas formas. Los biólogos definen polimorfismo como la variación en forma y función encontrada entre los miembros de especies comunes.
Por ejemplo, en una colonia de abejas, la reina es mucho más grande que cualquiera de las otras abejas. Su función especial es la propagación. Las otras abejas son típicamente pequeñas y tiene trabajos y funciones especializadas que varían desde la construcción de paneles hasta el traslado del polen. Aún las formas y órganos de las abejas varían, por ejemplo, las abejas transportadoras de polen tienen sacos en sus patas. Pero las abejas de una colonia son todas miembros de la misma especie, no obstante sus diferencias en formas y funciones. Por lo tanto, el termino polimorfismo connota las propiedades compartidas por una especie como las abejas así como las propiedades por las cuales los miembros de la especie difieren.
El polimorfismo requiere de la herencia. El polimorfismo simplifica el trabajo del programador al generalizar la sintaxis de la comunicación, la cual permitirá tratar objetos de diferente tipo de una manera similar. Esto es, el polimorfismo usa la herencia para expresar comúnmente en la forma de un protocolo de comunicación para envio de mensajes a objetos similares (pero no exactamente iguales).
Se ha establecido previamente que la programación orientada a objetos es programación de envío de mensajes a objetos de tipo desconocido. Dado que el tipo de un objeto no es conocido cuando se le envía un mensaje, la respuesta del objeto no puede ser predicha. La programación orientada a objetos involucra poner muchos diferentes objetos de desconocida pero similar tipo en una colección y entonces, activar los objetos en turno por envío a cada uno del mismo mensaje.
Por ejemplo, un vídeo juego podría consistir en una nave espacial, torpedos láser, naves enemigas, bombas, explosiones y paracaidistas. Cada uno de estos objetos comparten ciertas propiedades: Una posición y velocidad en la pantalla. Cada objeto tiene una forma o imagen que permite dibujarlos en la pantalla. Además de tener varias propiedades en común, cada uno de ellos comparte una conducta en común: Saben como moverse a la siguiente posición indicada por su velocidad, ellos saben como mostrarse por si mismos y pueden checar si su posición coincide con la posición de cualquier otro objeto en el juego. Tiene además un método destruir, el cual genera para la mayoría de los objetos una explosión y la eliminación de este objeto de la lista de elementos activos del juego.
Con el polimorfismo, a cada elemento en una colección se le envía el mismo mensaje, pero cada uno de ellos esta programado para reaccionar de diferente manera. Cada uno de los objetos en el vídeo juego comparte información básica y cada uno responde al conjunto de mensajes requeridos para simular el juego. Sin embargo, cada clase de objetos responderá un poco diferente a los mensajes

básicos -crear, destruir, chocar, aparecer y moverse. Por ejemplo, se requerirá de un método diferente para dibujar la nave espacial del usuario que para dibujar un fotón torpedo o un paracaidista. Pueden tomarse diferentes acciones ya sea que la posición de la nave espacial del usuario coincida o no con un torpedo enemigo o un paracaidista buscando rescate. Este es el polimorfismo en acción. El polimorfismo es la habilidad que tienen objetos similares de responder al mismo mensaje de diferentes maneras.

COMO EL POLIMORFISMO DIFIERE DE LA HERENCIA.

La herencia es por si misma simplemente una herramienta para reutilizar los códigos. A través de la herencia, los objetos pueden heredar todas o algunas de las características de otros objetos. La herencia también puede ser usada para evitar o restringir el acceso a rasgos proporcionados por la clase base. El polimorfismo se aplica solo a los métodos heredados de la clase base. El polimorfismo se requiere, además de la herencia para programar con orientación a objetos. La herencia es una herramienta mas general que el polimorfismo, es útil para modificar el comportamiento de una clase que provee funcionalidad cercana a la que se quiere, pero que no va a todas las nueve yardas.
Suponga que se tienen una clase Punto y una clase Rectángulo con funcionalidad limitada, y usted quiere añadir un Mouse para el movimiento de objetos en la pantalla. Para hacer esto usted necesita preguntar a cada objeto de la pantalla si ocupa la posición que el Mouse esta apuntando cuando se activa el botón del Mouse. Una técnica común es asignar una área rectángulo a cada figura y ventana, representado un hot spot que responde a la activación del botón del Mouse. En Smalltalk, se conoce como una caja de enlace. La colección de los objetos en la pantalla es un arreglo polimorfico. Esto es, los elementos dentro del arreglo tienen muchas diferentes formas o tipos, pero cada uno de ellos sabe como responder cuando se pregunta su retorno a la caja de enlace. La necesidad de añadir un Mouse es la manera de ver si el rectángulo (caja de enlace) contiene un punto especificado (el apuntador del Mouse) para cada elemento en la pantalla.
La clase original Rectángulo no provee este método. Sin embargo, todo lo que se necesita es derivar una nueva clase, llamada MiRectángulo que hereda todos los métodos de la clase base Rectángulo y añade un nuevo método, que contiene Punto.

```
class MyRectángulo : public Rectángulo
(
public :
    int containsPoint (Point p) ( /*...*/ )
)
```

Este es el más básico uso de la herencia -extender o restringir las capacidades de una clase existente. El polimorfismo usa la herencia para más propósitos - expresar comúnmente. Esto es, el polimorfismo usa la herencia específicamente para construir jerarquías de tipo polimorfico. Las jerarquías de tipo polimorfico son como las jerarquías normales , excepto que el protocolo de comunicación para todos los tipos de jerarquía polimorfica se establecen en la clase base para la jerarquía total. El polimorfismo requiere la definición de una clase base que especifica un conjunto de operaciones o procedimientos que todas las subclases usaran para comunicarse.

1.5 SOBRECARGA DE OPERADORES Y FUNCIONES

Algunos LOO, para mantener una total ortodoxia con el modelo de objetos, implementan las operaciones básicas, como las que tienen que ver con simple aritmética , con la filosofía del envío de mensajes a objetos. De esta manera sumar dos números i y j debería interpretarse considerándose a uno de los números como un objeto y enviándole el mensaje **sumar** con el otro número como argumento del mensaje. Si pensamos en una arquitectura de computadora convencional que incluye operaciones directas como las aritméticas esto es una sobrecarga de tiempo.

OPERADORES SOBRECARGADOS

El termino operador sobrecargado se refiere a un operador que es capaz de desarrollar su función en varios contextos diferentes, sin necesidad de otras operaciones adicionales. Por ejemplo, la suma a + b, en la práctica para nosotros supondrá operaciones diferentes dependiendo de que estemos trabajando en el campo de los números reales o en el campo de los números complejos. Si dotamos al operador + para que, además de sumar reales, sepa también sumar complejos, dependiendo esto del tipo de los operandos, entonces diremos que el operador + esta sobrecargado.

OPERADORES

La palabra clave operator mas un operador de los siguientes, forman el nombre de la función, el cual utilizaremos cada vez que necesitamos referenciarla al operador:

```
+   -   *   /   %   ^   &       |
-   |   .   =   <   >   <=      >=
++  --  <<  >>  ==  !=  &&      ||
-=  -=  *=  /=  %=  ^=  &=      !=
<<=  >>=  [ ]  ()  ->  new  delete
```

ARGUMENTOS

Es una lista de al menos un argumento, el cual tiene que ser un objeto de una clase específica. Quiere esto decir, que no están permitidos argumentos de tipos básicos.
Los operadores sobrecargados son normalmente utilizados con clases para facilitar las operaciones con un tipo de datos definido por el usuario. Los operadores sobrecargados: =, !!, ->, (), new y delete, deben ser definidos como funciones miembro de una clase. El resto de operadores no requiere esta exigencia.

SOBRECARGA DE FUNCIONES

Nos puede preocupar la eficiencia y queremos evitar las operaciones de conversión porque hacen perder tiempo. Esto puede lograrse dando otras definiciones para las funciones operator asignación (=), operator suma y asigna (+), operator suma (+) y operator comparación (= =) pero ahora con argumento de tipo int..
Por ejemplo en el lenguaje C++ permite que se use un mismo nombre de función en distintas clases. Esto no provoca ambigüedad porque las funciones sólo pueden llamarse a través de objetos de la clase. C++ también permite tener más de una definición para una función dentro de una misma clase . las diferentes definiciones de una misma función tienen que diferenciarse en la cantidad y/o en el tipo de los parámetros de manera que esto no genere ninguna posible situación ambigua.

1.6 GENERICIDAD Y TEMPLATES

En la programación de clases es frecuente encontrarnos con clases que son muy semejantes y que solo se diferencian en el tipo de algunas de las entidades de la clase (componentes, parámetros de algunas funciones de la clase, valor retornado por una función). Estas clases ofrecen una misma funcionalidad pero con parámetros o valores de retorno de tipo diferente.

CLASES CONTENEDORAS

Esta forma de reusabilidad paramétrizando clases es conocida como genericidad. Definiendo clases con parámetros que representen clases cualesquiera podemos evitar tener que reescribir clases muy parecidas que se diferencien sólo en el tipo de alguna de las entidades con las que se trabaja dentro de la clase.
La genericidad no es un recurso original de la POO pero es un arma que combinada con el modelo de objetos potencia la flexibilidad y la reusabilidad que éste promueve.

TEMPLATES Y CLASES GENERICAS

El concepto de *template* espresa la genericidad y define *clases paramétrizadas*.
A través de un *template* podemos definir clases (metaclases) que permitirán "generar" clases en tiempo de compilación de las que luego podrán ser generados objetos en tiempo de ejecución.

Parámetros genéricos de tipo

La construcción template class T especifica que a continuación sigue una definición genérica. El *argumento de tipo* (el identificador T en este caso) será utilizado en la definición que sigue a template class T . Esta clase genérica no puede ser utilizada directamente para crear objetos sino para "crear clases" asociando un tipo (*parámetro real genérico*) al *parámetro real genérico* T (que es considerado como si fuera de tipo "class").

Otros parámetros en templates

Aunque es la situación más usual, el argumento de un *template* no necesita ser el nombre de un tipo. Además de un nombre de tipo, el argumento de un *template* puede ser una cadena de caracteres, un nombre de una función o una expresión constante.

TEMPLATES Y FUNCIONES GENERICAS

Un *template* de función describe a una familia de funciones de la misma manera que un *template* de clase describir a una familia de clases. Puede ser útil definir clases genéricas que apliquen internamente a alguna componente del tipo del parámetro genérico una operación mas especifica que las anteriores, es decir, que utilizen una operación no aplicable a cualquier tipo que pueda asociársele al parámetro genérico, esto es conocido como genericidad restringida.

ARREGLOS Y TEMPLATES

Vectores

Uno de los usos más útiles de los *templates*, en combinación con la redefinición de operadores, es para mejorar el recurso interno (*built in*) de arreglos (*arrays*).
Podemos, por ejemplo, definir una clase genérica VECTOR de manera que cada objeto de tipo VECTOR tenga internamente su longitud (información que en general hay que estar manejando explícitamente y por separado cuando se trabaja con arreglos tradicionales.

GENERICIDAD Y HERENCIA

La genericidad es un recurso que tiene sentido en lenguajes tipados (*typed lenguajes*), es decir donde cada entidad se declara de un cierto tipo, de modo que sea posible determinar cuándo una operación es correcta desde el punto de vista del tipo.

GENERICIDAD Y TEMPLATES

En el mundo real se necesita con frecuencia que la definición de una clase o función, pueda ser utilizada con tipos u objetos diferentes, sin tener que reescribir (o realizar operaciones "cortar " y " pegar" , sustituyendo algunos elementos) el código varias veces.
Meyer ha definido genericidad de la siguiente forma:
"Genericidad es la capacidad para definir módulos parametrizados. Tal módulo, denominado módulo genérico, no es útil directamente; más bien es un patrón o modelo módulo. En la mayoría de los casos comunes, los parámetros significan tipos. Los módulos reales, llamados instancias del modulo genérico, se obtienen proporcionando tipos reales de cada uno de los parámetros genéricos."
Esta definición ha sido implementada en clases genéricas.
El propósito de la genericidad es definir una clase (o una función) sin especificar el tipo de uno o más de sus miembros (parámetros). Con este método se puede cambiar la clase para acoplar los usos diferentes sin tener que reescribir . A una clase de este tipo se denomina una clase contenedora. Supongamos que tiene que escribir el código para un conjunto de elementos (tal como una lista, una cola o una pila). Se puede definir un conjunto de enteros y utilizarlos. Si, posteriormente, tiene que definir también un conjunto de fechas, como se prosigue? Existen diferentes lenguajes que han solucionado este problema, tales como Object Pascal, C++, Small-talk, Eiffel, Clu, etc.
La traducción de template en español no es fácil: se pueden adaptar los términos plantilla, patrón, modelo, tipo genérico o incluso tipo parametrizado.
En particular, nosotros preferimos utilizar el termino plantilla, aunque una interpretación estricta del término puede exigir el uso del termino en inglés template.
Las plantillas se utilizan para definir clases y funciones con parámetros que normalmente son tipos. Las plantillas se utilizan para generación de clases y funciones ordinarias.

PLANTILLAS DE FUNCIONES

Una función de plantilla define una familia de funciones, cada una funcionando con un tipo de dato particular. Esta caracteristica en el caso de no existir plantillas en su compilador, se puede conseguir mediante sobrecarga de funciones, aunque con la restricción que impone la verificación estricta de tipos de C++ .

PLANTILLAS DE CLASES

Aunque las funciones genéricas son muy útiles, no son el uso más eficaz de las plantillas. Las clases genéricas son, realmente, el uso más empleado y adecuado.
Al igual que las plantillas de funciones nos permiten definir funciones cuyo tipo de argumento esta parametrizado, las plantillas de clases nos permiten definir clases genéricas que pueden manipular varios tipos de datos. Esas clases genéricas son, sobre todo, útiles para implementar contenedores que son clases que contiene objetos de un tipo dato: las clases contenedores permiten administrar listas enlazadas de objetos (ordenas o no), tablas cuyo tamaño puede variar dinámicamente, conjuntos, etc.
Dos nombres de clases plantilla se refieren a la misma clase solo si los nombres de plantillas son idénticos y sus argumentos tienen valores idénticos.
La implantación de clases contenedoras genéricas utiliza tres métodos: macros, herencia y plantillas. Las plantillas o patrones (templates) también llamados tipos genéricos o parametrizados, permiten construir una familia de funciones o clases relacionadas
Las plantillas de funciones define una familia de funciones parametrizadas.
Por ejemplo, se puede definir una función parametrizada ordenar para ordenar cualquier tipo de arreglo o lista, con el formato:

```
template <class  T> void ordenar (Array <T>)

   // cuerpo de la función ordenar
   // .....
}
```

Este formato declara un conjunto de funciones sobrecargadas ordenar, una para cada tipo de array. Cuando se define la función de ordenación, se necesita un operador de comparación de la clase T. Por consigulente, una restricción al utilizar la plantilla de función ordenar es que la clase que se proporciona en lugar de T , debe tener definido un operador de comparación.
Se puede invocar a la función ordenar como a cualquier función ordinaria.
Las plantillas le ayudan a definir clases que son generales por naturaleza (clases genéricas).
Las plantillas no son clases ordinarias y no se pueden pensar en ellas de igual forma que se piensa en herencia, pero ambas tienen conceptos comunes. La herencia y plantillas permiten crear nuevas clases a partir de una macro de trabajo existente; sin embargo, la herencia relaciona la nueva clase con las clases existentes, mientras que una plantilla es simplemente un medio de fabricar nuevas clases. De hecho, se puede pensar en template como la palabra reservada que soporta clases contenedoras.
Las plantillas son una característica extremadamente útil.

1.7 OBJETOS COMPARTIDOS

Especificación de componentes static

La declaración de una componente de un objeto puede estar precedida de la especificación **static**. Esto significa que la componente será *compartida* por todas las instancias de la clase. Es decir el espacio para la componente no se replica en cada objeto de la clase (en la terminología SMALLTALK estas componentes son llamadas *variables de clase* y las componentes no static *variables de instancia*).
De esta forma la clase puede incluir información general y no particular de cada objeto. Tal vez el ejemplo más utilizado para ilustrar este recurso es el de *contar* cuántas instancias tiene la clase.

```
class A
 { //...
   public:
     static int contador;
     A(...) { contador ++; //...Otras acciones propias del constructor
            }
   //...
   -A() { contador--; //...Otras acciones propias del destructor
        }
```

Todo constructor debe incluir ++ y todo destructor debe incluir contador --. De esta forma, como un objeto no puede crearse sin evaluar un constructor, se cuentan todos los objetos de la clase. Esta cuenta se decrementa cada vez que se destruye un objeto.
Para que la clase anterior se comporte *consistentemente* es necesario que la componente contador se *inicializada* antes de crear cualquier instancia de la clase. Esta *inicialización* es llevada a cabo por el compilador antes de comenzar la ejecución del programa (dar el control a la función main). El valor con el que será inicializada una componente **static** deberá especificarse fuera de la definición de la clase, es decir en la forma:

```
int A::contador=0;
```

Las componentes **static** que sean de tipo simple int son inicializadas a 0 por omisión.
Una aplicación, que tenga componentes **static** en varias clases, no debe depender del orden en que el compilador pueda llevar a cabo estas inicializaciones.

1.8 EXCEPCIONES

Las excepciones son condiciones no usuales en un programa. En general, son anomalías de programas en tiempo de ejecución tales como división por cero, desbordamiento o de rangos de arrays y agotamiento de almacenamiento de memoria libre.
Las excepciones pueden ser errores completos que producen fallos en el programa o condiciones que conducen a errores. Tipicamente, se pueden detectar siempre si ocurren ciertos tipos de errores. Por ejemplo, se puede detectar si los índices de un array están fuera de rango o si se trata de valores válidos. Tales errores se llama excepciones asincronas, debido a que aparecen en un momento previsible (un error producido por salirse el valor del índice fuera de rango, solo aparecerá después de la ejecución de una sentencia donde se utilice ese índice fuera de rango). En contraste con estas excepciones, existen otras denominadas excepciones asincronas que se producirán por sucesos fuera del control de su programa. Por consiguiente, su programa no puede anticipar su momento de aparición.
Con la excepción de los constructores y destructores, todas las funciones pueden devolver un valor. El tipo de valor se debe especificar en la declaración de la función. Sin embargo, si en el interior de una función de división de coma flotante se produce un problema, realmente no se necesita devolver un valor, sino mas bien otro objeto diferente tal como una cadena, por ejemplo desbordamiento de división.
Una de las características más notables de las excepciones en un lenguaje por ejemplo en el lenguaje C ++ es que las mismas pueden devolver cualquier tipo de objeto de una función, con independencia del tipo devuelto por la función. Otra caracterlstica de las excepciones es que si se ha construido alguno de los objetos que han sido construidos. Por consiguiente el tratamiento de excepciones es un medio alternativo para dejar un ámbito o alcance.
Una excepción se produce en el momento en que se produce una anomalía de un programa.
La excepción se maneja invocando a un manejador apropiado seleccionado de una lista de manejadores que se encuentran inmediatamente después del bloque del manejador.
Como en C++ no hay aseveraciones no puede aplicarse lingüísticamente la metáfora del contrato, La filosofía a aplicar es más sencilla y es conocida como dispara y captura (throw and catch). Una excepción ocurrirá aquí por dos posibles razones :
1. Una falla ocurrida y disparada en las capas bajas del lenguaje: hardware, sistema operativo, bibliotecas básicas del lenguaje.
2. Una excepción explícitamente programada (disparada) mediante una instrucción trow posiblemente al detectar algún incumplimiento de una condición .
La excepción disparada debe ser capturada en una sección catch que puede acompañar a la función que estaba ejecutándose cuando se disparo la excepción. Si no existe la tal sección catch la excepción se propagara a la función que llamo

a esta y así continuara elevándose hasta que alguna sección catch la capture o llegue a la superficie (salga de la función main) en cuyo caso el sistema de ejecución de C++ deberá reportar un error definitivo y abortar la ejecución.
Un elemento interesante que aporta C++ a la manipulación de excepciones es que al disparar una excepción se pueden pasar a objetos con ellas. Estos pueden contener información sobre la excepción y el contexto y situación en que ocurrió.
Complementariamente con esto podemos tener entonces diferentes secciones de captura según el tipo de sección disparada.
Una excepción disparada inicialmente por un throw sin parámetros, no puede ser capturada por nadie ni siquiera por una excepción catch del tipo catch . La utilidad de este tipo de throw es para pagar una excepción .
Una excepción no capturada se sigue propagando según el flujo de llamados hasta que alguien la capture (si es la capa más externa del sistema entonces se abortará la ejecución) . El problema con la filosofía del contratado honesto, que vimos anteriormente no puede garantizarse aquí porque después que se sale de una cláusula de captura C++ no obliga a reintentar la acción que originó la excepción ni informar a quien llamo (propagar la excepción). Es decir podemos salir silenciosamente de una cláusula de captura.
Una solución para cumplir con el principio de honestidad es que quien la capturo la vuelva a disparar . El inconveniente en comparación con dos lenguajes EIFFEL y C++ es que este nuevo disparo hay que programarlo explícitamente, es decir no es garantizado automáticamente por C++. Cuando se vuelve a disparar la excepción no hay que volver a pasar el objeto excepción como parámetro, C++ pasa el mismo parámetro.
Con programar en C++ un efecto equivalente al retry de EIFFEL. Un intento de solución es programar disciplinadamente involucrando siempre a la sección try a todo el cuerpo de la función y que la sección catch vuelva a llamar a la función .
Para usar las excepciones correctamente se necesita una estrategia general en la que varias partes de un programa se pongan de acuerdo en cuales excepciones y donde deben ser detectadas y como y por quienes deben ser tratadas. La filosofía de manejo de excepciones es inherente no local. Esto implica que esta estrategia debe considerarse desde las etapas iniciales del diseño de un sistema.
En definitiva las excepciones pueden ocurrir porque:

- La noción de correcto y confiable puede ser demasiado complicada para expresarla consistentemente las aseveraciones como las de EIFFEL ayudan pero no son semánticamente todo lo completas que pudiera necesitarse). Por ejemplo . las aseveraciones incluyen una lógica basada en el cálculo proposicional y no en el cálculo de predicados (no se incluyen cuantificadores).
- La sobrecarga en tiempo y espacio para hacer los chequeos necesarios de una programación preventiva es demasiado grande para que el sistema en su conjunto ejecute aceptablemente.
- Se utiliza hardware, recursos del sistema operativo y funciones de otros lenguajes que no obedecen estas reglas.

- Somos humanos y nos equivocamos.

Las excepciones han sido aceptadas por el comite ANSI C ++. Las excepciones son generalmente condiciones de error imprevistas. Normalmente estas condiciones terminan el programa de usuario con un mensaje de error proporcionado por el sistema .
El tratamiento de excepciones es la última característica importante que se ha añadido a ANSI C ++. Sin embargo esta caracteristica es dificil de implementar y casi ningún compilador comercial la tiene implementada. Las excepciones, cambiarán el modo de programar, así como el modo de pensar sobre programación.
C++ permite levantar directamente una excepción en un bloque try utilizando la expresión throw. La excepción se maneja invocando un manejador apropiado seleccionado de una lista que viene inmediatamente después del bloque try del manejador. Los menejadores de excepciones catch gestionan el tratamiento de excepciones .
Para usar las excepciones correctamente se necesita una estrategia general en la que varias partes de un programa estén diseñadas e interrelacionadas entre si, de modo que se deben decidir cuales son las excepciones, donde deben ser detectadas y como y por quienes deben ser tratadas.

1.9 Persistencia

Se dice que un objeto es persistente si continúa existiendo después de que la aplicación del programa ha finalizado. Tipicamente, los objetos que existen durante un programa orientado a objetos obtienen existencia en una de las tres formas:

1. El objeto es creado estáticamente o dinámicamente por el programa.

2. El objeto es obtenido de un archivo o una base de datos relacional y traducido a la forma de un objeto.

3. El objeto se recupera a partir de una base de datos orientada a objetos y ya esta listo para su uso.

Los objetos persistentes contienen datos que permanecen durante mas tiempo que el que emplea el programa de aplicación. A diferencia de las aplicaciones construidas a partir de lenguajes orientados a objetos, en las que los objetos se crean durante la ejecución y finalizan con la sesión de la aplicación, los objetos de una base de datos sobreviven múltiples sesiones. Asimismo, y a diferencia de los datos de los objetos persistentes no modifican su forma, aunque pueden moverse entre las aplicaciones y la base de datos en la que residen permanentemente.

La persistencia de objetos esta intimamente relacionada con el concepto de identidad de object, el cual requiere que cada objeto tenga algo acerca de lo que es único e invariable, no importando de como cambie el objeto. La identidad del objeto permite la manipulación eficiente de relaciones complejas. Si los objetos se refieren a cada uno los otros, por sus identificadores únicos de objeto, entonces sus relaciones continuarán ya sea como el cambio del estado de los objetos o su localización. Los contrastes de identidad de los objetos con bases de datos basadas en valor, tales como las bases de datos relacionales, donde las entidades son identificadas por sus atributos y que por lo tanto pueden cambiar con el tiempo. En otras palabras, la identidad de los objetos es especialmente útil en la persistencia de relaciones de objetos. Esta persistencia de objetos y su relación requiere de diferentes mecanismos para borrar los objetos, si un objeto en una base de datos orientada a objetos es borrada simplemente, otros objetos podrían ser dejados con referencias a los objetos borrados con que son ahora incorrectos. En lugar de esto, muchas bases de datos orientadas a objetos no borran objetos inmediatamente. En lugar de ello, la base de datos borra las referencias a un objeto borrado, el objeto borrado es removido cuando todas las referencias a este han sido removidas, y después el espacio es reclamable.

Los lenguajes de programación tradicional usualmente se dirigen hacia las tres primeras clases de persistencia de objetos; la persistencia de las tres ultimas clases es típicamente el dominio de la tecnología de Bases de Datos. Esto conduce al choque de culturas que en ocasiones se refleja en diseños muy extraños: programadores a la cabeza con esquemas ad hoc para almacenamiento de objetos cuyo estado debe ser preservado entre ejecuciones del programa, y diseñadores de bases de datos que aportan su tecnología para competir con objetos de transición.

La unificación de los conceptos de concurrencia y objetos conduce a los lenguajes de programación orientados objetos concurrentes. En forma similar, la introducción del concepto persistencia al modelo de objetos conduce a bases de datos construidas sobre tecnologías aprobadas, tales como una red secuencial, indexada, jerárquica o modelos de bases de datos relacionales, pero después de ofrecer al programador la abstracción de una interface orientada a objetos, a través de la cual los filtros de bases de datos y otras operaciones son complementadas en términos de objetos cuyo tiempo de vida trasciende de un programa individual. Esta unificación simplifica ampliamente el desarrollo de ciertas clases de aplicaciones.

La persistencia se encarga de mas de un tiempo de vida de datos. En bases de datos orientadas a objetos, no solo hace el estado de persistencia de un objeto, sino que su clase debe también trascender algún programa individual para que cada programa interprete éste estado salvado en la misma forma. Esto provoca claramente un desafío para mantener la integridad de una base de datos como si creciera, particularmente si deseamos cambiar la clase de un objeto.

La decisión pertenece a persistencia en el tiempo. En la mayoría de los sistemas, una vez creada consume la misma memoria física hasta que cesa de existir. Sin embargo, para sistemas que se ejecutan bajo un conjunto distribuido de

procesadores, debemos en ocasiones estar conscientes de la persistencia a través del espacio. En tales sistemas, es útil pensar en objetos que pueden moverse de máquina a máquina, y que pueden tener diferentes representaciones en diferentes máquinas.
Para sumarizar, se define a la persistencia como sigue:
" PERSISTENCIA es la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (por ejemplo, el objeto continua existiendo después de que su creador cesa de existir), y/o espacio por ejemplo, la ocasión del objeto se mueve del espacio de dirección en el cual fue creado. "
Como ya se había mencionado muchos de los lenguajes de programación orientados a objetos proporcionan la noción de objetos persistentes.
La idea de persistencia de un objeto conduce naturalmente a la esfera de bases de datos y al término más largo de almacenamiento de objetos. El paradigma orientado a objetos ofrece algunas poderosas herramientas para describir las estructuras y las relaciones las cuales son explotadas en las bases de datos orientadas a objetos.
Las bases de datos abren la posibilidad de almacenar objetos no solo entre diferentes aplicaciones. Por lo tanto llega a ser necesario para las clases, así como para los objetos, llega a ser "persistente" para que el estado de representación de objetos pueda ser interpretada correctamente en diferentes programas.

PORQUE ES NECESARIA LA PERSISTENCIA

Los programas de computación utilizan memoria volátil para su tiempo de corrida en el almacenamiento s=datos. Esto se aplica de igual en programas de C++ y COBOL. Ambos objetos creados por programas de C++ y registros creados por programas de COBOL pueden ser transferidos para respaldar el almacenamiento si ellos son persistentes.

FORMAS DE LOGRAR LA PERSISTENCIA

Hoy en día, los usos mas importantes en la persistencia de objetos es negociar con el hueco existente entre programas orientados objetos y mecanismos de almacenamiento convencional. A continuación se sugieren cuatro formas de cruzar estos huecos.
1. Saltar hueco. La estrategia del saltado del hueco es una aproximación ad hoc para los programas orientados a objetos deseando almacenar objetos en un archivo o base de datos. Para cada clase de objetos, el programador idea un mapeado en la representación plana e implementa el mapeado como un par de funciones. Un simple mapeado puede ser para bases de datos relacional hasta el contenido de una tabla para cada clase de objetos que necesite ser almacenada. Cada clase puede ser referenciada por otra clase definida por cualquier llave única y las referencias interobjeto son representadas por medio del uso de esa llave. Las recuperaciones especiales de los programas pueden ser un código duro

dentro de rutinas asociadas con cada clase. Arbitrariamente las cuestiones podrían no ser soportadas. Mapear objetos dentro de una base de datos involucra un proceso no trivial de complejos aplastamientos y posibles estructuras de datos recursivas dentro del límite de los tipos de estructuras de las bases de datos. El mismo desajuste ocurre en reversa cuando la lectura de datos esta dentro del programa. El problema es además componer las diferencias entre lenguajes declarativos, como SQL, y un lenguaje de programación. Aún así, esta estrategia puede producir una solución práctica y una muy eficiente interface para un sistema de archivos o bases de datos, y en el costo del esfuerzo de programar. Hoy en día la estrategia "salto de huecos" es una de las más utilizadas por muchos programadores orientados a objetos, cuando ellos quieren almacenar objetos o comunicarse entre programas. La analogía es fuerte: muchos huecos pueden ser saltados, pero uno desea tener que hacer esto muy frecuentemente o cuando se carga con basura dura (como la herencia o el polimorfismo).

2. Construcción de un puente. Esta opción es una generalización de la estrategia "saltar de hueco". Una capa de interface elegante es construida que transparentemente conecta programas escritos en lenguajes de programación orientados a objetos para bases de datos convencionales, posiblemente utilizando sentencias estándar SQL. La capa de interface o puente, define un algoritmo general para trazar un mapa de objetos para representaciones de bases de datos y otro respaldo.
Muchos intentos para construir un puente parecido se han enfocado en el uso de bases de datos relacionales como medios de almacenamiento (por ejemplo. la versión del prototipo IRIS y el trabajo de Cooke). El puente luce muy parecido a una base de datos orientada a objetos en el inicio y fin mientras se utiliza un manejador de bases de datos relacional en el otro fin. Mientras nosotros creamos que es como un puente podría soportar todas las construcciones orientadas a objetos. esto es nuestra experiencia en el desempeño puede ser pobre debido a que a las estructuras de almacenamiento les hace falta ser optimizadas para los diferentes tipos de acceso requerido por el modelo orientados a objetos.
En teoría, el tener datos en el fin del respaldo relacional es semejante a un esquema el cual puede ser accesible para los programas no orientados a objetos. En la práctica, las estructuras de almacenamiento de datos pueden tender a ser antinaturales y no convenientes de utilizar en este sentido.
Para construir la transparencia del puente se requiere programación en el nivel metaobjeto, debido a que los algoritmos para convertir la representación son lógicamente asociadas con las clases antes que las instancias en el sistema de objetos. Lenguajes como Smalltalk y CLOS son ideales. Lenguajes como C++, el cual no soporta este tipo de programación hace esto mucho más difícil el proveer un puente transparente.
Existen posibles confusiones entre los puentes descritos aquí y el creciente número de librerías, ahora accesible para una variedad de lenguajes orientados a objetos, que proporcionan interfaces para relacionar a otras de datos. El mejor conocimiento de todas estas librerías prevén solamente un nivel de soporte bajo,

dando simple acceso a las características de la base de datos. Ellos no convienen con el mapeo sistemático entre objetos y registros, en los cuales permanece la responsabilidad de el programador.

3. Hueco Angosto. La estrategia de la realización de la base de datos relacional es una estrategia evolucionaría más que revolucionaria de las nuevas compañías de bases de datos orientadas a objetos. Ellas están extendiendo gradualmente la estructura de los datos que las bases de datos relacionales incluyen, o bien están imitando, características de lenguajes orientados a objetos. Su objetivo no es construir un puente pero realizar esto es innecesario para reducir progresivamente el hueco entre la interface de bases de datos y el modelo de programación.
La ventaja de esta estrategia es generalizar y compatibilizar con datos existentes. Desde los desarrollos evolucionados no hacen invalidas las bases de datos previas, programas existentes y los datos pueden coexistir con unos nuevos. La tecnología relacional ha gastado muchos anos de desarrollo con mecanismos sofisticados para la optimización de querys y de la confianza de la seguridad, ventajas que pueden ser conservadas en algún desarrollo evolucionado. En el lado contrario, se puede sentir inseguridad a acerca del alcance del abismo entre el modelo relacional y el modelo orientado a objetos pudiendo ser disminuidos los sobrantes mientras están dentro los límites de la teoría relacional. Ahí se presentan algunas preguntas. Si bien algunos vendedores relacionales claman que en teoría ellos pueden hacer que optimicen las operaciones de los queries tan rápido como los siguientes puntos para navegación de queries, en la práctica las bases de datos orientadas a objetos se ve que pueden ser mucho más rápidas. Además, de que los programadores pueden continuar programando en dos lenguajes, por ejemplo SQL, y un lenguaje de propósito general.
La tercera generación de los sistemas manifiestos de base de datos, describe la posible evolución de bases de datos desde el punto de vista de la comunidad de bases de datos relacional.

4. Llenar el vacío. La última estrategia es ignorar la tecnología de bases de datos actual y construir completamente un sistema administrador de almacenamiento que mapee eficientemente los objetos de la memoria principal ubicados en la memoria de la base de datos y su respaldo, guardando intacta su estructura inherente. La base de datos es optimizada para soportar eficientemente las características de los lenguajes de programación orientados a objetos. Por ejemplo, datos en disco pueden ser agrupados de acuerdo a los patrones de acceso pero esta recuperación en objetos es deseada para recuperar y ocultar muchos de los objetos accesibles de esos objetos en una sola recuperación.
Este camino es uno de los más elegidos por la mayoría de los vendedores corrientes disponibles de bases de datos orientadas a objetos. Estos productos son frecuentemente integrados muy fuertemente con un lenguaje simple de programación, comúnmente C++, y proporciona un concepto no simple para accesar a los objetos de cualquier otro lenguaje.

Un importante beneficio demandado para los programas orientados a objetos es la flexibilidad. El encapsulamiento y el ocultamiento de la información permiten la representación de objetos para ser cambiados con un impacto mínimo.
Nuevos acuerdos o cambios de requerimientos pueden guiar a la reorganización de la jerarquía de la clase. Para soportar esta flexibilidad, la idea general o esquema de la estructura de la clase sostenida en la base de datos tiene la tarea de cambiar en la misma línea. Objetos de bases de datos corrientes proveen solo mecanismos límites para un esquema de evolución. Sin embargo, en muchos casos las bases de datos imponen la definición de las clases de objetos en la base de datos que muchos corresponden exactamente a la clase definida en todos los programas accesados a la base de datos.
A pesar de sus diferencias, los objetos corrientes de las bases de datos pueden dar conceptos substanciales y representaciones adelantadas sobre bases de datos relacionales para aplicaciones durante las cuales estas restricciones no convienen y así son encontradas y leídas en un mercado.
Añadir persistencia a un objeto, es decir, como hacerlo durar más allá que el programa que lo creó, es un problema importante dentro de la POO. Supongamos que tenemos una clase persistente que tiene dos funciones salva y carga mediante las cuáles se pueden hacer copias y cargas de un objeto en disco. Toda clase A que quiera añadir estos recursos a sus objetos tendría que heredar de persistente sin que esto impida que pueda también heredar de otras clases.
Este recurso, de añadir funcionalidad a una clase por la vía de la herencia múltiple facilita que una instrumentación incluya clases predefinidas en bibliotecas para resolver problemas sin tener que introducir modificaciones en el lenguaje.

2. BASES DE DATOS RELACIONALES

2.0 INTRODUCCION

Las bases de datos relacionales aparecieron a finales de los años 70 y ofrecían varias ventajas sobre el modelo jerárquico. Los métodos para manipular los datos en las tablas son bien conocidos y razonablemente eficaces. El álgebra relacional proporciona una base matemática formal capaz de soportar las metodologías de análisis de datos, Por último, con la posibilidad de unir dinámicamente los datos en tablas virtuales durante la ejecución, las bases de datos relacionales facilitaron el desarrollo de aplicaciones con un nivel más alto de independencia de datos que sus predecesoras.
Sin embargo, el modelo relacional sufre al menos de un inconveniente notable. Es difícil expresar la semántica de objetos complejos con sólo un modelo de tabla para el almacenamiento de datos. Aunque las bases de datos relacionales son adecuadas para contabilidad u otras aplicaciones típicas de procesamiento de transacciones, en las que los tipos de datos son simples y escasos en número, el modelo relacional ofrece ayuda limitada cuando los tipos se vuelven numerosos y complejos.
Al igual que el paso de jerárquica a relacional fue dado para reducir la atención hacia la estructura, también lo hizo así el paso de relacional a orientada a objetos. Las nuevas aplicaciones de los años 90 requieren soporte para estructuras de datos extensibles y complejas, acceso a datos complejos y alto rendimiento. Los conceptos tradicionales de bases de datos se están ampliando gradualmente con las capacidades orientadas a objetos para satisfacer estas necesidades. La evolución de la tecnología de bases de datos durante los últimos 20 años, representa este giro hacia una gestión de datos mas complejos.

2.1 TABLAS

El principio de organización de una base de datos relacional es la tabla, una disposición rectangular fila/columna de los valores de datos. Cada tabla de una base de datos tiene un nombre de tabla único que identifica sus contenidos. En realidad, cada usuario puede elegir sus propios nombres de tablas sin preocuparse acerca de los nombres elegidos por otros usuarios.
También es importante conocer que el conjunto de valores que una columna puede contener se denomina dominio de la columna.
Las columnas de una tabla tienen un orden de izquierda a derecha, que se define cuando la tabla se crea por primera vez. Una tabla siempre tiene al menos una columna. El estándar SQL ANSI/ISO no especifica un número

máximo de columnas en una tabla, pero casi todos los productos comerciales SQL imponen un límite que generalmente es de 255 columnas por tabla o mas.
A diferencia de las columnas, las filas de una tabla no tienen orden particular. De hecho, si se utilizan dos consultas de bases de datos consecutivos para visualizar los contenidos de una tabla no hay garantía de que las filas sean listadas en el mismo orden dos veces. Naturalmente se puede solicitar a SQL que ordene las filas antes de visualizarlas, pero el orden de clasificación no tiene nada que ver con la disposición efectiva de las filas dentro de la tabla.
Una tabla puede tener cualquier número de filas. Una tabla de cero filas es perfectamente legal, y se le denomina una tabla vacía (por razones obvias). Una tabla vacía sigue teniendo una estructura, impuesta por sus columnas; simplemente no contiene datos. El estándar ANSI/ISO no limita el numero de filas de una tabla, y muchos productos SQL permiten que una tabla crezca hasta que agote el espacio de disco disponible en el computador. Otros productos SQL imponen un límite máximo, pero este es siempre muy generoso dos millones de filas o mas en común.

Claves Primarias

Puesto que las filas de una tabla relacional no están ordenadas, no se puede seleccionar una fila específica por su posición en la tabla. No hay "primera fila" , "ultima fila", o "decimotercera fila" de una tabla.
En una base de datos relacional bien diseñada cada tabla tiene una columna o combinación de columnas cuyos valores identifican unívocamente cada fila en la tabla. Esta columna (o columnas) se denomina clave primaria de la tabla.
La clave primaria tiene un valor único diferente para cada fila de una tabla, de modo que no hay dos filas de una tabla con clave primaria que sean duplicados exactos la una de la otra. Una tabla en donde cada fila es diferente de todas se llama una relación en términos matemáticos. El nombre base de datos relacional proviene de este termino ya que las relaciones (las tablas con filas distintas) son el corazón de una base de datos relacional.

Relaciones

Una de las principales diferencias entre el modelo relacional y los modelos de datos primitivos es que los punteros explícitos, tales como las relaciones padre/hijo de una base de datos jerárquicos, están prohibidas en las bases de datos relacionales. Obviamente estas relaciones siguen existiendo en una base de datos relacional.

Claves Foráneas

Una columna de una tabla cuyo valor coincide con la clave primaria de alguna otra tabla se denomina una clave foránea.

Juntas, una clave primaria y una clave foránea crean una relación padre/hijo entre las tablas que las contienen, del mismo modo que las relaciones padre/hijo de una base de datos jerárquicos.
Lo mismo que una combinación de columnas puede servir como clave primaria de una tabla, una clave foránea puede ser también una combinación de columnas. De hecho, la clave foránea será siempre una clave compuesta (multicolumna) cuando referencia a una tabla con una clave primaria compuesta.
Obviamente el número de columnas y los tipos de datos de las columnas en la clave foránea y en la clave primaria deben ser idénticos unos a otros.
Las claves foráneas son parte fundamental del modelo relacional ya que crean relaciones entre tablas en la base de datos. Desgraciadamente, como con las claves foráneas, el soporte de claves foráneas falta en los sistemas de gestión de base de datos relacional.

Las doce reglas de Codd

En su artículo de 1985 en computerworld, Ted Codd presento doce reglas que una base de datos debe obedecer para que sea considerada verdaderamente relacional. Las reglas se derivan del trabajo teórico de Codd sobre el modelo relacional, y representan realmente mas un objetivo ideal que una definición de una base de datos relacional.
Ningún DBMS relacional actualmente disponible satisface totalmente las doce reglas de Codd. De hecho, se esta convirtiendo en una practica popular elaborar "tarjetas de tanteo" para productos DBMS comerciales, que muestran lo bien o mal que estos cada una de las reglas. Desgraciadamente, las reglas son subjetivas de modo que los calificadores están generalmente llenos de notas de ple de página y calificadores, y no revelan demasiado acerca de los productos.

- La Regla 1 es básicamente la definición informal de una base de datos relacional. Porque expresa que toda la información de una base de datos relacional esta representada explícitamente a nivel lógico y exactamente de un modo mediante valores en tablas.

- La Regla 2 refuerza la importancia de las claves primarias para localizar datos en la base de datos. El nombre de la tabla localiza la tabla correcta, el nombre de la columna encuentra la columna correcta y el valor de la clave primaria encuentra la fila que contiene un dato individual de interés.

- La Regla 3 requiere soporte para falta de datos mediante el uso de valores NULL..
  Esto quiere decir que los valores nulos distintos de la cadena de caracteres vacía o de una cadena de caracteres en blanco y distinta del cero de cualquier otro número se soportan en los DBMS completamente relacionales

para representar la falta de información y la información inaplicable de un modo sistemático e independiente del tipo de datos.

- La Regla 4 requiere que una base de datos relacional sea autodescriptiva. En otras palabras, la base de datos debe contener ciertas tablas de sistema cuyas columnas describan la estructura de la propia base de datos.

- La Regla 5 ordena la utilización de un lenguaje de base de datos relacional, tal como SQL, aunque no se requiera especificamente SQL. El lenguaje debe ser capaz de soportar todas las funciones básicas de un DBMS creación de un base de datos, recuperación y entrada de datos, implementación de la seguridad de la base de datos, etc.

- La Regla 6 trata de las vistas, que son tablas virtuales utilizadas para dar a diferentes usuarios de una base de datos diferentes vistas de su estructura. Es una de las reglas mas difíciles de implementar en la práctica, y ningún producto comercial la satisface totalmente hoy dia.

- La Regla 7 refuerza la naturaleza orientada a conjuntos de una base de datos relacional. requiere que las filas sean tratadas como conjuntos en operaciones de inserción, supresión y actualización. La regla esta diseñada para prohibir implementaciones que solo soportan la modificación o recorrido fila a fila de la base de datos.

- La Regla 8 y la Regla 9 aislan al usuario o al programa de aplicación de la implementacion de bajo nivel de la base de datos. Especifican que las técnicas de acceso a almacenamiento especificadas y utilizadas por el DBMS, e incluso los cambios a la estructura de las tablas en la base de datos, no deberían afectar a la capacidad del usuario de trabajar con los datos.

- La regla 10 dice que el lenguaje de base de datos debería soportar las restricciones de integridad que restringen los datos que pueden ser introducidos en la base de datos y las modificaciones que puedan ser efectuadas en esta. Esta es otra de las reglas que no soportan la mayoría de los productos comerciales DBMS.

- La Regla 11 dice que el lenguaje de base de datos debe ser capaz de manipular datos distribuidos localizados en otros sistemas informaticos; es decir que un DBMS relacional tiene independencia de distribución.

- La Regla 12 impide "otros caminos" en la base de datos que pudieran subvertir su estructura relacional y su integridad; más explícitamente podemos decir que si un sistema relacional tiene un lenguaje de bajo nivel (un sólo registro cada vez), ese bajo nivel no puede ser utilizado para

subvertir o suprimir las reglas de integridad y las restricciones expresadas en el lenguaje relacional de nivel superior (múltiples registros a la vez).

SQL esta basado en el modelo relacional de datos que organiza los datos en una base de datos como una colección de tablas:

- Cada tabla tiene un nombre que la identifica unívocamente.
- Cada tabla tiene una o mas columnas nominadas, que están dispuestas en un orden especifico de izquierda a derecha.
- Cada tabla tiene cero o más filas, conteniendo cada una un único valor en cada columna. Las filas están desordenadas.
- Todos los valores de una columna determinada tienen el mismo tipo de datos, y estos están extraídos de un conjunto de valores legales llamado el dominio de la columna.

Las tablas están relacionadas unas con otras por los datos que contienen. El modelo de datos relacional utiliza claves primarias y claves foráneas para representar estas relaciones entre tablas:

- Una clave primaria es una columna o combinación de columnas dentro de una tabla cuyo(s) valor(es) identifica(n) unívocamente a cada fila de la tabla. Una tabla tiene una única clave primaria.

- Una clave foránea es una columna o combinación de columnas en una tabla cuyo(s) valor(es) es(son) un valor de clave primaria para alguna otra tabla . Una tabla puede contener mas de una clave foránea, enlazándola a una o mas tablas.

- Una combinación clave primaria/clave foránea crea una relación padre/hijo entre las tablas que la contienen.

Nombres de tabla
Cuando se especifica un nombre de tabla en una sentencia SQL, SQL presupone que se esta refiriendo a una de las tablas propias (es decir una tabla ya creada).
Con el permiso adecuado, también se puede referir a tablas propiedad de otros usuarios, utilizando un nombre de tabla cualificado. Un nombre de tabla cualificado. especifica el nombre del propietario de la tabla junto con el nombre de la tabla, separados por un punto(.). Por ejemplo, la tabla CUMPLEAÑOS, propiedad del usuario de nombre SAM, tiene el nombre de tabla calificado:
Un nombre de tabla cualificado puede ser utilizado generalmente dentro de una sentencia SQL en cualquier lugar que pueda aparecer un nombre de tabla.

Nombres de columna
Cuando se especifica un nombre de columna en una sentencia SQL, SQL puede determinar normalmente a que columna se refiere a partir del contexto. Sin embargo, si la sentencia afecta a dos columnas con el mismo nombre correspondientes a dos tablas diferentes, debe utilizarse un nombre de columna cualificado para identificar sin ambigüedad la columna designada. Un nombre de columna cualificada especifica tanto el nombre de la tabla que contiene la columna como el nombre de la columna, separados por un punto (.). Por ejemplo, la columna de nombre VENTAS en la tabla REPVENTAS tiene el nombre de columna cualificado.

REPVENTAS .VENTAS

Si la columna procede de una tabla propiedad de otro usuario, se utiliza un nombre de tabla cualificado en el nombre de columna cualificado. Por ejemplo, la columna DIA_MES en la tabla CUMPLEAÑOS propiedad del usuario SAM se especifica mediante el nombre de columna cualificado completo:

SAM.CUMPLEANOS.DIA.MES

Los nombres de columna cualificados pueden ser utilizados generalmente en una sentencia SQL en cualquier lugar en el que pueda aparecer un nombre de columna simple (no cualificado); las excepciones se indican en las descripciones de las sentencias SQL individuales.

2.2 TIPOS DE DATOS

El estándar SQL ANSI/ISO especifica varios tipos de datos que pueden ser almacenados en una base de datos basada en SQL y manipulados por el lenguaje SQL: Los tipos de datos especificados por el estándar son sólo un conjunto mínimo, pero casi todos los productos SQL comerciales, los soportan o tienen tipos de datos que son muy similares. Los tipos de datos ANSI/ISO consisten en los siguientes:

- Cadenas de caracteres de longitud fija. Las columnas que contienen este tipo de datos almacenan típicamente nombres de personas y empresas, direcciones, descripciones, etc.

- Enteros. Las columnas que contienen este tipo de datos almacenan típicamente cuentas, cantidades, edades, etc. Las columnas de valores enteros también se utilizan con frecuencia para contener números identificadores, tales como números de cliente, de empleado y de pedido.

- Números decimales. Las columnas con este tipo de datos almacenan números que tienen parte fraccionaria y deben ser calculados exactamente,

tales como porcentajes y tasas. También se utilizan frecuentemente para almacenar importes monetarios.

- Números en coma flotante. Las columnas con este tipo de datos se utilizan para almacenar números científicos que pueden ser calculados aproximadamente, tales como pesos y distancias. Los números en coma flotante pueden representar mayores rangos de valores que los números decimales, pero pueden producir errores de redondeo en los cálculos.

Tipos de datos extendidos
La mayoría de los productos SQL comerciales ofrecen un conjunto mucho mas extenso de tipos de datos que los especificados en el estándar ANSI/ISO.
Algunos de los tipos de datos extendidos más populares o importantes son:

- Cadenas de caracteres de longitud variable. Casi todos los productos SQL soportan los datos VARCHAR, que permiten que una columna almacene cadenas de caracteres que varían de longitud de una fila a otra, hasta una cierta longitud máxima. El estándar ANSI/ISO solamente especifica cadenas de longitud fija, que son rellenadas por la derecha con caracteres en blanco adicionales.

- Importes monetarios. Muchos productos SQL soportan un tipo MONEY o CURRENCY, que se almacena generalmente como número decimal o en coma flotante. El tener un tipo monetario distinto permite al DBMS formatear adecuadamente los importes monetarios cuando son visualizados.

- Fechas y horas. Soportar valores para fechas y horas es también habitual en los productos SQL, aunque los detalles varían ampliamente de un producto a otro. Generalmente se soportan variadas combinaciones de fechas, horas, instantes, intervalos de tiempo y aritmética de fecha y hora.

- Datos Boleados. Algunos productos SQL (incluyendo dBASE IV) soportan los valores lógicos (True o False) como un tipo explícito.

- Texto extenso. Varias bases de datos basadas en SQL soportan columnas que almacenan cadenas de texto extensas (típicamente de hasta 32000 o 65000 caracteres, o entre 10 o 20 páginas escritas a un solo espacio). El DBMS restringe generalmente el uso de estas columnas en consultas y búsquedas interactivas.

- Flujos de bytes no estructurados. Oracle y otros varios productos permiten almacenar y recuperar secuencias de bytes de longitud variable sin estructurar.

Las columnas que contienen estos datos se utilizan para almacenar imágenes de vídeo comprimidas, código ejecutable y otros tipos de datos sin estructurar,

- Caracteres asiáticos. DB2 soporta cadenas de longitud fija y de longitud variable de caracteres de 16 bits utilizados para representar caracteres Kanji y otros caracteres asiáticos. Sin embargo, la búsqueda y ordenación de estos tipos GRAPHIC y VARGRAPHIC no esta permitida.

2.3 TIPOS ESTRUCTURADOS Y BLOBS (Imágenes y Sonidos - Multimedia)

La principal limitación de SQL como un lenguaje de Bases de Datos relacionales es su incapacidad para manejar datos complejos dentro de las columnas de una tabla. El atributo teléfono(s) en una tabla de personas puede ser de contenido variable. Hay personas que tendrán varios números telefónicos y otras que no tendrán ninguno. Una extensión a SQL puede hacer que un campo contenga cero, uno, dos y hasta cualquier cantidad de ocurrencias definiéndolo como un conjunto de datos elementales.
De la misma forma una columna podría contener arreglos de una o más dimensiones o incluso tablas completas. Esto desde luego rompe con el modelo original de Bases de Datos Relacionales, respecto a que en cada campo sólo puede haber datos atómicos.
Por otro lado, dentro de una tupla o registro podemos necesitar, además de datos elementales (números, cadenas de caracteres, fechas, etc.) almacenar imágenes en dos o tres dimensiones, o fotografías.
También puede requerirse almacenar sonidos (voz, música instrumental, canto, ruidos, etc.).
Tanto las imágenes como el sonido pueden digitalizarse y representarse como información binaria. Sin embargo, la longitud de esta información puede ser considerable. Estamos hablando de varios miles o millones de bits según la complejidad y detalle de las imágenes o la fidelidad y duración del sonido.
Esta información puede almacenarse como archivos independientes o bien como campos de una tupla o registro. Debido a su tamaño relativamente grande (100,000 o más bits) respecto a los datos elementales (de 1 a 80 bits), se les ha denominado Objetos Binarios Largos (Binary Long Objects y su abreviación BLOB).
Los BLOBs pueden almacenarse en disco de distintas maneras. Dbase maneja un tipo especial de campos denominados MEMO; los cuales almacena en archivos separados (un archivo por cada tipo de campo MEMO) del resto de la definición del registro o renglón de una tabla relacional el cual almacena en un archivo DBF.
Una forma alterna de almacenamiento de los BLOBs seria junto con el resto de la definición del registro; esto es; un sólo archivo físico correspondiendo a cada

tabla lógica. Finalmente, si los BLOBS son demasiado grandes se puede crear un archivo separado para cada ocurrencia de ese campo.
Si los BLOBs son imágenes para la pantalla, pueden insertarse como componentes de alguna forma, junto con otros campos del mismo registro (p.e. la foto y la firma de una persona junto con su nombre, dirección, teléfono, etc.); Esto ya lo hacen muchos paquetes comerciales como Foxpro. Lo que ya es más difícil es tratar de recuperar estas imágenes en forma múltiple (varios registros) sin relacionarlas con el resto del registro (por ejemplo mostrar todas las fotos de personas con cabello negro).
Si los BLOBs contienen información acústica (voz, música, clave morse, alarma), puede reproducirse (amplificador y bocinas) al tiempo que se despliega en la pantalla el resto de los campos del registro; o bien, reproducirse únicamente el sonido de varios registros sin el resto de campos.

2.4 LENGUAJE DE CONSULTA SQL

Una diferencia fundamental entre sistemas de bases de datos relacionales y orientadas a objetos radica en la cantidad de información que se mueve entre la aplicación y el sistema de gestión de la base de datos. Las aplicaciones envían consultas a la base de datos relacional, la cual devuelve entonces un numero de valores. Estos valores son almacenados normalmente como parte de las estructuras de datos de la aplicación, manipulados, y vueltos a entregar a la base de datos para su almacenamiento. En contraste con ello, cuando las aplicaciones envían mensajes a las bases de datos orientadas a objetos, la base de datos manipula los datos con los métodos, recupera o calcula un valor, y devuelve el valor a la aplicación.
Existe mucho más tráfico entre una aplicación y una base de datos relacional que almacena objetos. En una base de datos relacional de ese tipo, los objetos complejos son divididos y almacenados como campos en tablas independientes, de forma que un número de consultas debe aplicarse para recuperar y volver a ensamblar un solo objeto. Estas consultas se aplican secuencialmente, de forma que una consulta depende del resultado de la consulta anterior. La base de datos no conoce la petición global; conoce solamente las consultas individuales y por consiguiente no puede reordenar las consultas para optimizarlas. En una base de datos totalmente orientada a objetos, un solo mensaje toma el lugar de muchas consultas de base de datos relacional. Un mensaje puede solicitar cálculos y hacer que se envíen mensajes a otros objetos antes de entregar un resultado. Esto alivia el problema de la necesidad de consultas secuenciales.
A medida que SQL se ha convertido en un estándar oficioso en las consultas de bases de datos relacionales, las bases de datos orientadas a objetos han sido forzadas a incorporarle. Y ello a pesar del hecho de que la estructura relacional y SQL no son consistentes con las estructuras de una base de datos orientada a objetos. Las consultas según el predicado relacional se dirigen contra

estructuras desconocidas, pero la consulta orientada a objetos navega a través de punteros que definen una estructura conocida.
Para resolver estas diferencias, las bases de datos orientadas a objetos utilizan dos métodos principales. Uno de ellos es ampliar los tipos reconocidos por la sintaxis de SQL, para permitir que sean consultados los objetos y también las tablas. Otro método es permitir las consultas SQL y las del objeto en tandem y posteriormente unir las dos consultas en la aplicación. El problema principal con las extensiones orientadas a objetos para SQL ha sido el rendimiento; el proceso de coincidencia y búsqueda relacional es intrInsecamente lento cuando se le compara con una base de datos puramente orientada a objetos en la que las consultas referencian directamente los objetos.
SQL es una herramienta para organizar, gestionar y recuperar datos almacenados en una base de datos informática. El nombre SQL es una abreviatura de Structured Query Languaje (Lenguaje de Estructuras de Consultas) . Por razones históricas ,. SQL se pronuncia generalmente :"sequel", pero también se emplea la pronunciación alternativa "SQL". Como su nombre implica, SQL es un lenguaje informatico que se puede utilizar para interaccionar con una base de datos . En efecto, SQL trabaja con un tipo especIfico de base de datos, llamada base de datos relacional.
El programa informatico que controla la base de datos se denomina sistema de gestión de base de datos (database management system) o DBMS.
Cuando se necesita recuperar datos de la base de datos, se utiliza el lenguaje SQL para efectuar la petición. El DBMS procesa la petición SQL, recupera los datos solicitados y los devuelve. Este proceso de solicitar datos de la base de datos y de recibir los resultados se denomina consulta (query) a la base de datos de aquí el nombre Structured Query Language. El nombre Structured Query Language es realmente y en cierta medida inapropiado. En primer lugar, SQL es mucho más que una herramienta de consulta, aunque ese fue su propósito original, además recuperar sigue siendo una de sus funciones mas importantes, SQL se utiliza para controlar todas las funciones que un DBMS proporciona a sus usuarios, incluyendo:

- Definición de Datos. SQL permite a un usuario definir la estructura y organización de los datos almacenados y de las relaciones entre ellos.

- Recuperación de datos. SQL permite a un usuario o a un programa de aplicación recuperar los datos almacenados de la base de datos y utilizarlos.

- Manipulación de datos. SQL permite a un usuario o a un programa de aplicación actualizar la base de datos añadiendo nuevos datos, suprimiendo datos antiguos y modificando datos previamente almacenados.

- Control de acceso. SQL puede ser utilizado para restringir la capacidad de un usuario, para recuperar, añadir y modificar datos, protegiendo así los datos almacenados frente a accesos no autorizados.

- Comparición de datos. SQL se utiliza para coordinar la comparición de datos por parte de usuarios concurrentes, asegurando que no interfieren unos con otros.

- Integridad de datos. SQL define restricciones de integridad en la base de datos, protegiéndola contra corrupciones debidas a actualizaciones inconsistentes o a fallos del sistema.

Por tanto SQL es un lenguaje completo de control e interactuacion con un sistema de gestión de base de datos.
Finalmente, SQL no es un lenguaje particularmente estructurado, especialmente cuando se compara con lenguajes altamente estructurados tales como C o Pascal. En vez de ello, las sentencias SQL se asemejan a frases en ingles, que se completan con palabras de relleno que no añaden nada al significado de la frase pero que hace que se lea más naturalmente. Hay unas cuantas inconsistencias en el lenguaje SQL, y también existen algunas reglas especiales para impedir la construcción de sentencias SQL que parecen perfectamente legales, pero que no tienen sentido.
A pesar de la imprecisión de su nombre, SQL ha emergido como el lenguaje estándar para la utilización de bases de datos relacionales. SQL es a la vez un potente lenguaje y un lenguaje relativamente fácil de aprender.
SQL no es en si mismo un sistema de gestión de base de datos, ni un producto autónomo. Usted no puede ir a una tienda de informática y comprar SQL. En su lugar, SQL es parte integral de un sistema de gestión de datos, un lenguaje y una herramienta para comunicase con el DBMS.
La maquina de base de datos es el corazón del DBMS, responsable de estructurar, almacenar y recuperar realmente los datos en el disco. Acepta peticiones SQL procedentes de otros componentes DBMS, tales como una facilidad de formularios, un escritor de informes o una facilidad de consultas interactivas, desde otros sistemas informaticos.

- SQL es un lenguaje de consultas interactivas. Los usuarios escriben ordenes SQL en un programa SQL interactivo para recuperar datos y mostrarlos en la pantalla, proporcionando una herramienta conveniente y fácil de utilizar para consultas ad hoc de la base de datos.

- SQL es un lenguaje de programación de base de datos. Los programadores insertan ordenes SQL en sus programas de aplicación para acceder a los datos de la base. Tanto los programas escritos por el usuario como los programas de utilidad de la base de datos (tales como los escritores de informe y las herramientas de entrada de datos) utilizan esta técnica para acceso a la base de datos.

- SQL es un lenguaje de administración de base de datos. El administrador de la base de datos es responsable de gestionar una base de datos en un

minicopmutador o en un maxicomputador, utilizando SQL para definir la estructura de la base de datos y para controlar el acceso a los datos almacenados.

- SQL es un lenguaje cliente/servidor. Los programas de computador personal utilizan SQL para comunicarse sobre una red de área local con servidores de base de datos que almacenan los datos compartidos. Muchas aplicaciones novedosas están utilizando esta arquitectura de cliente/servidor, que minimiza el tráfico por la red y permite que tanto las PCs como los servidores efectúen mejor su trabajo.

- SQL es un lenguaje de base de datos distribuidos. Los sistemas de gestión de base de datos distribuidos utilizan SQL para ayudar a distribuir datos a través de muchos sistemas informaticos conectados. El software DBMS de cada sistema utiliza SQL para comunicarse con los otros sistemas, enviando peticiones para acceso a datos.

- SQL es un lenguaje de pasarela de base de datos. En una red informática con una mezcla de diferentes productos DBMS. SQL se utiliza a menudo en una pasarela (gateway) que permite que nuestro producto DBMS se comunique con otro producto.

SQL ha emergido por tanto como una herramienta potente y útil para enlazar personas y sistemas informaticos a los datos almacenados en una base de datos relacional.

Características y beneficios de SQL.
SQL es a la vez un lenguaje fácil de entender y una herramienta completa para gestionar datos. He aquí algunas de las principales características de SQL y las fuerzas del mercado que le han hecho tener éxito:

- Su independencia de los vendedores.
- Su portabilidad a través de sistemas informaticos.
- Los estándares SQL.
- El apoyo de IBM.
- Su fundamento relacional.
- Su estructura de alto nivel semejante al ingles.
- Las consultas interactivas ad hoc.
- Su acceso a la base de datos mediante programas.
- Las vistas mutiples de datos.
- El ser un lenguaje de base de datos.
- Su definición dinámica de datos.
- La arquitectura cliente/servidor.

Estas son las razones por las que SQL ha emergido como la herramienta estándar para gestionar datos en computadores personales, minicomputadores y maxicomputadores. Cada una de ellas se describen a continuación:

- Interdependencia de los vendedores.
  SQL es ofertado por todos los principales vendedores de DBMS y ningún producto nuevo de base de datos puede tener éxito sin el soporte de SQL. Una base de datos basada en SQL y los programas que la utilizan pueden transferirse de un DBMS al DBMS de otro vendedor con mínimo esfuerzo de conversión y poco reentrenamiento del personal. Herramientas de base de datos basadas en SQL que funcionan con varios productos de DBMS, tales como escritores de informe y generadores de aplicación, están comenzando a aparecer. La independencia del vendedor proporcionada así por SQL es una de las razones mas importantes de su popularidad.

- Portabilidad a través de sistemas informaticos.
  Los Vendedores de DBMS en SQL ofertan sus productos sobre sistemas informaticos que van desde computadores personales y estaciones de trabajo hasta redes de área local, minicomputadores y maxicomputadores. Las aplicaciones basadas en SQL que comienzan en sistemas monousuario pueden ser transferidas a sistemas mayores de minicomputadoras o maxicomputadoras cuando crecen. Los datos procedentes de bases de datos corporativas basadas en SQL pueden ser extraídas y remitidas a bases de datos departamentales o personales. Finalmente, los económicos computadores personales pueden ser utilizados para construir prototipos de aplicaciones de bases de datos basadas en SQL antes de transferirlas a un sistema multiusuario costoso.

- Estándares SQL
  El American National Standards Institute (ANSI) y la International Standards Organización (ISO) han publicado conjuntamente un estándar oficial para SQL. SQL se ha convertido también en un estándar de U.S. Federal Información Processing Standard (FIPS)., lo que le convierte en un requerimiento esencial para los grandes contratos informaticos del gobierno. En Europa, X/OPEN, un estándar para un entorno de aplicación por cable basado en UNIIX, ha añadido SQL como el estándar para acceso a base de datos. La Open Software Foundation (OSF), un grupo de vendedores de UNIX, planea basar su estándar de acceso a base de datos en SQL. Estos estándares sirven como sello oficial de aprobación para SQL y han acelerado su aceptación en el mercado.

- Apoyo de IBM
  SQL fue inventado originalmente por investigadores de IBM, y desde entonces se ha convertido en un producto estratégico para IBM. SQL es un componente esencial de la Systems Aplication Architecture (SAA), la marca

de IBM para la compatibilidad de sus diversas lineas de productos. El soporte SQL esta disponible en todas las cuatro familias de sistemas incluidas bajo el paraguas SAA: los computadores personales PS/2, los sistemas de medio rango AS/400 y los maxicomputadores de IBM que ejecutan los sistemas operativos MVS y VM. Este extenso soporte de IBM ha acelerado la aceptación del mercado de SQL, y proporcionado una clara señal de las intenciones de IBM para que otros vendedores de sistemas y bases de datos las sigan.

- Fundamento Relacional
  SQL es un lenguaje para base de datos relacional, y se ha popularizado juntamente con el modelo de base de datos relacional. La estructura tabular, de filas y columnas de una base de datos relacional, es intuitiva para los usuarios, y hace que el lenguaje SQL se mantenga simple y fácil de entender. El modelo relacional tiene también un fuerte fundamento teórico que ha guiado la evolución y la implementacion de las bases de datos relacionales. Cabalgando sobre una ola de aceptación provocada por el éxito del modelo relacional. SQL se ha convertido en el lenguaje de base de datos relacionales.

- Estructura de alto nivel en ingles
  Las sentencias SQL parecen sencillas frases en ingles, lo que hace que SQL sea fácil de aprender y entender: Esto es en parte debido a que las sentencias de SQL describen los datos a recuperar, en lugar de especificar como hallar los datos. Las tablas y columnas de una base de datos SQL pueden tener nombres largos y descriptivos. Como consecuencia, la mayoría de las sentencias SQL dicen lo que significan, y pueden ser leídas como frases claras y naturales.

- Consultas Interactivas ad hoc
  SQL es un lenguaje de consultas interactivas que proporciona a los usuarios acceso ad hoc a los datos almacenados. Utilizando SQL interactivamente, un usuario puede obtener respuestas incluso a cuestiones complejas en minutos o segundos, en fuerte contraste con los días o semanas que le llevaría a un programador escribir un programa de informe a medida. Debido a la potencia de consulta ad hoc de SQL, los datos son mas accesibles y pueden ser utilizados para ayudar a una organización a tomar decisiones mejores y mas informadas.
  Así podemos decir que la principal operación solicitada por los usuarios y sus aplicaciones es la consulta de la información previamente almacenada. Tomando en consideración el gran éxito logrado por lenguajes de consulta como SQL, es evidente la necesidad de continuar ofreciendo un lenguaje de consulta, que tomando en consideración el modelo OO, permita extraer objetos o parte de ellos a través de un lenguaje sencillo, de tipo declarativo.

- Acceso a la base de datos mediante programas
  SQL es también un lenguaje de base de datos utilizado por los programadores para escribir aplicaciones que acceden a una base de datos. Las mismas sentencias SQL se utilizan para acceso interactivo y luego insertadas dentro del programa. En contraste, las bases de datos tradicionales proporcionan un conjunto de herramientas para acceso mediante programas y una facilidad de consulta aparte para peticiones ad hoc, sin ninguna energía entre los dos modos de acceso.

- Vistas múltiples de datos
  Utilizando SQL, el creador de una base de datos puede dar a diferentes usuarios de la base de datos vistas diferentes de su estructura y contenidos. Por ejemplo, la base de datos puede ser construida de modo que cada usuario solo vea datos de su propio departamento o de su propia región de ventas. Además, los datos procedentes de diferentes partes de la base de datos pueden combinarse y presentarse al usuario como una simple fila/columna de una tabla. Las vistas de SQL pueden ser utilizadas de este modo para mejorar la seguridad de una base de datos y para acomodarla a las necesidades particulares de los usuarios individuales.

- Lenguaje completo de base de datos
  SQL fue inicialmente desarrollado como un lenguaje de consulta ad hoc, pero su potencia va ahora mas allá de la recuperación de datos. SQL proporciona un lenguaje complejo y consistente para crear una base de datos, gestionar su seguridad. actualizar sus contenidos, recuperar los datos y compartirlos entre muchos usuarios concurrentes. Los conceptos de SQL que son aprendidos en una parte del lenguaje, pueden ser aplicados a otras ordenes de SQL, haciendo mas productivos a sus usuarios.

- Definición dinámica de datos
  Utilizando SQL, la estructura de una base de datos puede ser modificada y ampliada dinámicamente, incluso mientras los usuarios están accediendo a los contenidos de la base de datos. Este es un avance importante sobre los lenguajes de definición de datos estáticos, que impiden el acceso a la base de datos mientras su estructura esta siendo modificada. SQL proporciona de este modo máxima flexibilidad, permitiendo que una base de datos se adapte a exigencias cambiantes mientras continúan sin ser interrumpidas las aplicaciones en línea.

- Arquitectura cliente/servidor
  SQL es un vehículo natural para implementar aplicaciones utilizando una arquitectura cliente/servidor distribuida: En este papel, SQL sirve como enlace entre los sistemas informaticos "frontales" (front-end) optimizados

para interacción con el usuario y los sistemas "de apoyo" (back-end) especializados para gestión de bases de datos, permitiendo que cada sistema rinda lo mejor posible. SQL también permite que los computadores personales funcionen como frontales de bases de datos mayores, dispuestas en minicomputadores y maxicomputadores, proporcionando acceso a datos corporativos desde aplicaciones informáticas personales.

3. ALMACENAMIENTO DE OBJETOS EN UNA BASE DE DATOS

3.0 INTRODUCCION

La persona que realiza un programa no necesita buscar a través de múltiples archivos por medio de punteros para determinar un dato específico. Además de los datos, los objetos pueden almacenar relaciones, representadas internamente como enlaces a otros objetos, y pueden almacenar el comportamiento, representado internamente como métodos.

Los objetos pueden hacer también que las aplicaciones se ejecuten mas rápido. Como los datos de una entidad están relacionados lógicamente, la base de datos orientada a objetos tiene los medios para optimizar su localización física. Las aplicaciones son capaces de leer menos archivos para recuperar todos los datos relevantes. Los sistemas tradicionales CAD, por ejemplo, almacenan a menudo cada componente de un diseño en un archivo separado y utilizan una base de datos para albergar los nombres de los archivos. Para acceder al diseño completo, la aplicación CAD debe abrir y cerrar todos los archivos de la lista de archivos. En una base orientada a objetos, el diseño entero con todos sus componentes puede almacenarse en un objeto, reduciendo grandemente el número de operaciones de archivo necesarias para acceder al diseño.

Las bases de datos orientadas a objetos proporcionan almacenamiento central para datos, mecanismos para compartir dichos datos entre los usuarios y formas de asegurar la integridad y seguridad de los datos. Recientemente, la tecnología de bases de datos ha luchado para encontrar formas de continuar proporcionando estas ventajas y al mismo tiempo incrementar el almacenamiento de datos complejos y diversos.

Siguiendo de cerca la pista de las extensiones de objetos para los lenguajes de programación, las bases de datos orientadas a objetos prometen aportar los mismos beneficios al mundo de la gestión de datos que los lenguajes orientados a objetos aportan al mundo de la programación. Al igual que en la programación orientada a objetos, las bases de datos orientadas a objetos intentan reducir la complejidad aumentando la capacidad de los fundamentos básicos del software. Las bases de datos orientadas a objetos y las extensiones orientadas a objetos para las tecnologías de bases de datos existentes ofrecen no solo formas de almacenar y recuperar datos sino también los mecanismos para definir y gestionar las complejas relaciones entre los datos. Las bases de datos orientadas a objetos no almacenan datos aisladamente sino más bien siguen el paradigma orientado a objetos de vincular los datos junto con el comportamiento asociado dentro de los objetos. Las capacidades funcionales de las bases de datos orientadas a objetos incluyen soporte de construcciones ricas en modelado de datos, soporte directo de Inferencia o deducción, y la posibilidad de almacenar tipos de datos, como por ejemplo, imágenes, audio o voz y vídeo. Las bases de datos orientadas a objetos van mas allá de la simple representación de datos como números y textos

pasivos; en su lugar modelan con mayor exactitud el mundo con su riqueza y textura.

Las bases de datos fueron construidas inicialmente para soportar aplicaciones que procesaran grandes volúmenes de datos uniformemente estructurados con una pequeña estructura interna o sin ella. Con la llegada en los años 70 de las bases de datos jerárquicas, se separaron las aplicaciones y los datos. Los modelos de datos satisfactorios desarrollados para las aplicaciones de los años 70 eran similarmente uniformes y simples en estructura. Los modelos eran con frecuencia orientados a objetos, con un único elemento de datos en cada campo. Este simple modelo, conocido como modelo jerárquico, permitía el desarrollo de funciones, tales como lenguajes de consulta, optimización de consultas, comprobación de limitaciones y control sobre la gestión de almacenamiento subyacente.

Las aplicaciones de inteligencia artificial (AI), como los sistemas expertos, guían también la necesidad del desarrollo de bases de datos orientadas a objetos. Los sistemas expertos se caracterizan por la necesidad de manejar muchos modelos de tipos de datos diferentes y a menudo complejos. Las bases de datos orientadas a objetos, en contraste con las bases de datos relacionales, son capaces de tratar estos requisitos sin sufrir degradación cuando aumenta el número de tipos de datos.

Llevadas por estas diversas necesidades de aplicación, a finales de los años 80, comenzaron a aparecer bases de datos orientadas a objetos, unas pocas comerciales y muchas basadas en investigación. La mayoría de estas bases de datos orientadas a objetos iniciales eran utilizadas en aplicaciones basadas en diseño en los campos científico y de la ingeniería.

Las primeras bases de datos iniciales orientadas a objetos fueron por ejemplo G-Base de Graphel, Vbase de Ontologic y GemStone de Servio Logic. G-Base, presentada en 1987, estaba basada en Lisp, con un lenguaje de desarrollo de aplicaciones de sistemas expertos íntimamente acoplado, (G-Logis).

Estas bases de datos orientadas a objetos iniciales tenían varias características que limitaban su empleo en el área general comercial. En primer lugar, la orientación hacia el diseño suponía que el usuario ejecutaba un número limitado de transacciones ampliadas, en comparación con las transacciones frecuentes y de gran volumen de muchas aplicaciones comerciales. En segundo lugar, los usuarios de ingeniería, al igual que los programadores, accedían a los objetos a través de un examen, mientras que los usuarios comerciales requerían herramientas de consulta fáciles de utilizar como las que se proporcionaban con el Lenguaje de Consulta Estructurado **SQL** (Structured Query Language).

Además, los intentos de los vendedores propietarios de bases de datos orientadas a objetos para proporcionar nuevas definiciones de bases de datos y lenguajes de manipulación en el sector comercial no tuvieron tampoco éxito, ya que SQL se había convertido en el estándar del área relacional. El temprano fracaso de Ontologic con COP es uno de esos ejemplos. Finalmente, estas realizaciones iniciales adolecían de bajo rendimiento, haciéndolas inadecuadas para aplicaciones a gran escala y de gran volumen. Claramente, el centro de las aplicaciones de diseño daba como resultado una disparidad entre la funcionalidad

del usuario, rendimiento y necesidades de optimización de las bases de datos comerciales.
Muchas de esas limitaciones se han intentado corregir con nuevos productos y/o mejoras a las Bases de Datos Orientadas a Objetos iniciales, como Ontos de Ontologic, han intentado corregir muchas de esas limitaciones. Ontos, por ejemplo, esta escrito en C++ y esta diseñado para ser abierto a los compiladores y herramientas de desarrollo de otros vendedores. El rendimiento está también mejorado en las versiones más recientes por medio de una variedad de métodos que incluye compiladores que analizan los flujos de trabajo para optimizar la posición de los objetos en la memoria, cache o almacenamiento del sistema.
Otro avance reciente en las bases de datos orientadas a objetos ha sido la extensión de la cualidad de hibridas de las bases de datos relacionales existentes para acomodar buena parte de la funcionalidad de una base de datos pura orientada a objetos. Las extensiones de objetos para el modelo relacional centran la optimización para las aplicaciones comerciales en el rendimiento, la representación física de los objetos para tratar con un entorno heterogéneo y distribuido, y la ampliación del modelo SQL para incluir el paradigma orientado a objetos. Estos esfuerzos son evidentes en el trabajo actual de los vendedores establecidos de bases de datos relacionales. Estos vendedores incluyen a INGRES Corporation con la base de datos INGRES Object Management Intelligent, a IBM Corporation con su deposito basado en DB2, a Hewlett-Packard con IRIS y a Oracle con su base de datos Oracle.
Estas extensiones del modelo de base de datos relacional para soportar objetos han tomado uno de los métodos principales. Un método es permitir que los mismos objetos actúen como tablas relacionales. En este método se crea una capa que permite a los objetos actuar como tablas. La capa descompone las estructuras objeto y las reduce para que puedan ser almacenadas en tablas relacionales. Los objetos actúan entonces como tablas virtuales y se les llama cuando se les necesita, utilizando un lenguaje estándar de consulta, como SQL. Estas capas almacenan información sobre los datos como objetos, pero los datos en si permanecen en tablas relacionales. La capa almacena y recupera solamente objetos completos.

3.1 BASES DE DATOS ORIENTADAS A OBJETOS PURAS

Una de las caracteristicas distinguibles de una base de datos de objeto es el soporte para un modelo de objeto . Se ha visto que éste modelo de objetos es importante porque permite determinar las clases de información que pueden ser guardadas en una base de datos de objeto y la semántica del objeto DBMS puede entenderse y mejorarse.

EL MODELO DE OBJETO ODMG SEPARA LA INTERFACE Y LA IMPLEMENTACION

El interior de un objeto es privado al objeto, mientras que el exterior es público y disponible para ser usado por otros objetos. El exterior es llamado la interface del objeto . El interior es llamado su implementación. La interface de un objeto es una especificación abstracta y no determina como se representa o implementa un objeto. Un desarrollador puede definir una interface de objetos, de la tecnología en particular, por ejemplo los lenguajes de programación o base de datos eventualmente usadas para implementar el objeto . Cada interface de los objetos describe sus características visibles . Visible aquí significa que son utilizables por alguien, una aplicación o un objeto externos al objeto.
La noción de interfase-implementación es esencial para el fundarnento conceptual de encapsulación de objetos. Ningún otro objeto puede accesar directamente la implementación de un objeto. Todos los accesos deben ser a través de la interface. La separación de la interface y la implementación es importante , ya que permite detallar la implementación de un objeto que cambia sin afectar otros objetos que usa el objeto. La semántica básica de los elementos del modelado de objetos en ODMG:

- El modelamiento fundamental primitivo es el objeto. El término objeto e instancia son usados intercambiablemente .
- Cada objeto es único identificable por un identificador de objeto también conocido como objeto - Id , el cual no se puede cambiar durante la vida del objeto.
- Los objetos son categorizados dentro de una jerarquía de tipos y subtipos . Todos los objetos de un tipo dado tienen características comunes de estado y comportamiento . Un subtipo hereda las características de su supertipo (supertype (s)).
- El estado es definido por los valores del objeto que llevan a un conjunto de propiedades. Estas propiedades pueden ser atributos de un objeto o relaciones entre el objeto y uno o más de otros objetos .
- El comportamiento es definido por el conjunto de operaciones que puede ser aplicada a un objeto .

No todos los modelos de objetos han sido semánticos. Por ejemplo C++ define un objeto como una región de almacenamiento este es un punto de vista de implementación de un objeto . En contraste, el modelo de objeto ODMG define un objeto como una abstracción, el cual puede ser implementado de varias maneras. De manera diferente los niveles de estructuras de implementación de datos y los métodos de la definición de C++ de clases, las propiedades y operaciones de los ODMG del modelo de objetos son especificaciones de interface . Cuando el modelo de objetos ODMG es "unido a " C++ para implementación, las propiedades y operaciones son representadas en C++ como datos de estructuras y métodos respectivamente. El modelo de objetos ODMG y el modelo de objetos

de la programación de los lenguajes de objetos son diferentes, pero deben trabajar juntos .

C++ TIENE SU PROPIO MODELO DE OBJETOS

Consideremos primero algunos fundamentos del modelo de objetos C++. Un objeto es una instancia de un objeto. Un programador define una clase especificando sus elementos de datos, por ejemplo el nombre, el tipo y sus funciones de miembros, cada uno con nombre y tipos de sus fundamentos , y su tipo de retorno. C++ soporta subclases, un conjunto predefinido de tipos de datos simples por ejemplo: CHAR, INT, SHORT, LONG, FLOAT, DOUBLE, un conjunto predefinido de tipos de estructuras por ejemplo, ARRAY y la habilidad de los usuarios para definir nuevas clases de estos agregados de elementos de datos y sus operaciones.
C++ es referido algunas veces como un lenguaje " estrictamente escrito" . Cada variable C++ es definida como un tipo particular. El compilador C++ checa todas las asignaciones de los valores para asegurar que todos los valores estén correctamente establecidos a los tipos de las variables deseadas. El checar los tipos se considera como una característica atractiva de C++ porque el chequeo prevé ciertos tipos de errores. Cuando ocurre un error de no correspondencia el compilador ya sea que aplique un tipo de conversión a la regla para forzar el valor de un tipo correcto o bien saca una bandera de error .
Las restricciones de mundo real determinan los tipos de variables.

Ejemplos de estas restricciones incluyen los siguiente:

- La variable Iterator debe ser del tipo INT, porque las iteraciones son contadas en número total de incrementos únicamente.
- La variable Document_Title debe ser del tipo String porque los nombres de documentos consisten de cadenas de textos.
- La variable Salary debe contener un valor numérico porque éste tipo de compensaciones es medida utilizando números y se calcula usando la variable como sea posible.
- El objeto empleado deben ser los elementos de datos y una función de miembros definida para la clase Empleado, porque el punto de vista de C++ es de que esto significa que es un empleado.

Cada restricción es determinada por su significado, esto es por la semántica de la variable del objeto. La ubicación de estas restricciones puede conducir a errores y fallas del programa. Lo atractivo de esta fuerza de escritura es basada en la suposición que más de un sistema puede forzar la semántica, lo más parecido es que los programas deben trabajar como se intento. Un beneficio es que el sistema pueda forzar la semántica disminuyendo la lógica del desarrollador que ha construido el código en el programa. Y menos lógica implica programas más

simples "el cual implica" "mas programadores productivos", el cual implica que el "software llegue al mercado más rápidamente", lo cual es bueno.

EL MODELO DE OBJETO ODMG DISTRIBUYE ENTRE VARIABLES Y CONSTANTES DE OBJETOS.

Consideremos ahora los detalles de algunos de los aspectos del modelo de objetos ODMG. Todos los objetos tienen características de comportamiento y estado. El comportamiento es definido por el conjunto de operaciones que puede ser aplicada para el objeto. El estado es definido por los valores que el objeto tiene para un conjunto de propiedades. Estas propiedades pueden ser atributo del objeto o relaciones entre el objeto y uno o más de otros objetos.
Hay dos tipos básicos de objetos mutables e inmutables. Mutable significa simplemente cambiable, inmutable significa constante y no cambiable. Los valores de los objetos mutables y de sus propiedades pueden cambiar, pero unas propiedades de objetos inmutables nunca cambian los objetos inmutables, y son comúnmente llamados "literales" y los objetos mutables son simplemente llamados "objetos". Ambos mutables e inmutables tienen todas las características de los objetos: identidad, tipo, estado, comportamiento, y otros . Para mantenerlo con esta convención, el término de objeto denotable es usado cuando se necesita utilizar un término más simple para referirse a ambos objetos. El objetivo denotable es usado en el sentido de objetos que son identificables o distinguibles .
La mutalidad es una restricción semántica importante. Si el objeto DBMS sabe que un objeto particular es inmutable, puede prohibir cualquier aplicación que haga cambios en el objeto. Por ejemplo un objeto que es un conjunto de 50 abreviaciones es inmutable. Las aplicaciones no deben cambiar las abreviaturas.
El conocimiento de la mutalidad también puede ser importante cuando se requiere determinar la implementación apropiada para la estrategia de un objeto. Por ejemplo los lenguajes de programación de objetos típicamente implementan literales que de alguna manera son diferentes que los objetos mutables.

TODOS LOS OBJETOS TIENEN IDENTIDAD UNICA

Todos los objetos denotables, tienen una identidad inmutable única. Cada objeto tiene una existencia separada y puede ser distinguido de otros objetos. Cuando un objeto es creado, el "sistema" asigna un identificador al objeto comúnmente referido como su objeto_id. El objeto DBMS usa un identificador para identificar un objeto, únicamente y probar su igualdad con otros objetos Dos objetos son el mismo si y sólo si tienen el mismo valor de identificador .
No todos los modelos incluyen la noción de igualdad basada en los identificadores de objeto.

EXISTEN MUCHAS MANERAS DE IMPLEMENTAR IDENTIFICADORES DE OBJETOS

La representación física de los identificadores de objeto no esta prescrita por el modelo del objeto. Los modelos DBMS de objetos que implementan el modelo del objeto escogen su propia estrategia de implementación. Un objeto DBMS puede utilizar diferentes aproximaciones para representar identificadores de objetos y literales . Una literal de identificador de objeto es usualmente representada por el patrón bit con código del valor de la literal, mientras que el identificador de un objeto mutable es representado por un patrón de bit único generado solamente para ese proceso.
Porque hay muchas maneras de implementar los identificadores de objeto, puede ser difícil, para paquetes de software de múltiples vendedores para compartir objetos. Por ejemplo, si dos objetos de DBMS utilizan diferentes esquemas para representar identificadores de objeto, ninguno pude accesar los objetos creados por el otro excepto que la misma translación de técnica esté disponible para los dos.

LOS OBJETOS PUEDEN TENER NOMBRES ASIGNADOS

Un objeto puede dar uno o más significados de aplicación en adición a su identificador de nombre . Un nombre es un valor literal string que puede ser utilizado por el objeto DBMS, para localizar un objeto. Estos nombres hacen fácil la aplicación de accesos a objetos de particular interés. Por ejemplo, un objeto que representa la geometría de la parte num. I23 puede ser nombrado "Part_I23_Geometry" . Un objeto que representa el documento num. X3H7-93-007v3 puede ser nombrado "X3H7-93-007v3 ". El mismo objeto podría también ser llamado de otra manera "X3H7 Modelo del objeto con aplicaciones de matriz". La aplicación determina el uso apropiado de los nombres .
Los nombres de los objetos no son los mismos como llaves primarias en modelos relacionales. Ellos difieren en dos cosas el escenario y la mutabilidad. Una llave primaria relacional es el conjunto de columnas de una tabla, los valores de los cuales están únicamente identificados por renglones en la tabla . El escenario de llaves primarias únicas, es la tabla en la cual está definida. Usando una llave primaria para localizar en renglones requiere reconocer cual del renglón está en la tabla. Las llaves primarias no son mutables si un valor participante en la llave primaria de un renglón cambia, el DBMS y las aplicaciones consideradas en el renglón deben ser diferentes ahora. Un objeto, sin embargo, puede ser asignado con nombres diferentes, y estos nombres pueden ser cambiados sin modificar la identidad o existencia del objeto.
La asignación y el significado de nombres de objetos es una aplicación de responsabilidad. El objeto de DBMS proporciona la capacidad de asignar nombres y de localizar los objetos con su nombre, pero no prevé aplicaciones para cambios de nombres en los cuales otras aplicaciones puedan tomar lugar el

objeto de DBMS; esencialmente proporciona un directorio de nombres para servicio de aplicaciones y para sus propios propósitos.

UN OBJETO TIENE CARACTERISTICAS, PROPIEDADES Y OPERACIONES

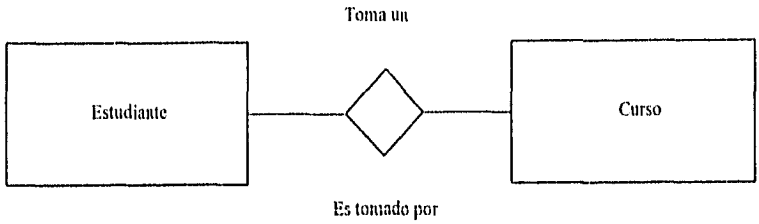
Las características de un objeto son un conjunto de propiedades y operaciones. Estas características son precisamente lo que describe la interface del objeto:

I. Las atribuciones para las cuales las aplicaciones pueden obtener un conjunto de valores y/o .
2. Las relaciones y las aplicaciones que pueden transversar de un objeto a otros
3. Las operaciones y aplicaciones que pueden adherirse a este objeto.

Algunos otros modelos de objeto, tales como OMG, CORBA y el SmallTalk, incluyen solo operaciones de interface en los objetos. El modelo de objetos ODMG incluye ambas operaciones y propiedades, en parte porque la influencia de la tecnología de base de datos en la cual tiene otros extremos y ha incorporado solo propiedades y no operaciones en los modelos de base de datos tradicionales, por ejemplo, consideremos una persona objeto. Tres características de este objeto son nombre, estatura y edad. El modelo de objetos ODMG podría considerar nombre, estatura y edad como atributos, es decir propiedades del tipo Persona. En contraste el modelo OMG y CORBA podría considerar nombre , estatura y edad como operaciones definidas de una parte de la interface de objetos de tipo Persona. El modelo de objeto ODMG deja abierta la implementación de unión de estos detalles tales como ya sea nombre, estatura y edad estos atributos son representados como datos de archivos o como funciones aplicadas para otros valores; como por ejemplo la fecha de nacimiento almacenados en el estado interno del objeto.

UNA RELACION INVOLUCRA A UNO O DOS OBJETOS

El segundo tipo de características de un tipo objeto son las relaciones.
Las relaciones están definidas en tipos pero no en propiedades de tipo- nivel, porque este tipo de instancias que participan en las relaciones no son del tipo mismo. Cualquier objeto puede participar en relaciones con otros objetos . Las relaciones del modelo de objeto ODMG no son las mismas correspondientes, aún cuando un puntero va también de un objeto a otro. Una relación es una abstracción que representa una asociación, entre objetos, mientras un apuntador es una construcción física.
Una relación es modelada por pares de relaciones asignadas, cada tipo de definición de los otros objetos involucrados en la relación y en el nombre de una trayectoria transversal usada para referir a los objetos relacionados; por ejemplo la siguiente figura muestra que un estudiante toma un conjunto de cursos y un curso es_tomado_por un conjunto de estudiantes.



RELACIONES ENTRE ESTUDIANTE Y TIPOS DE CURSOS

Los nombres de las trayectorias transversales están declarados dentro de la definición de interface de los tipos de objeto participantes en la relación . La trayectoria transversal también podría ser definida en la especificación de interface para el tipo Estudiante; la trayectoria transversal es_tomado _por, podría ser definida en la especificación de interface para el tipo curso. Estas trayectorias transversales son declaras como inversas en cada otro lado. Las trayectorias transversales cardinales están incluidas en la especificación del objetivo de la trayectoria transversal.

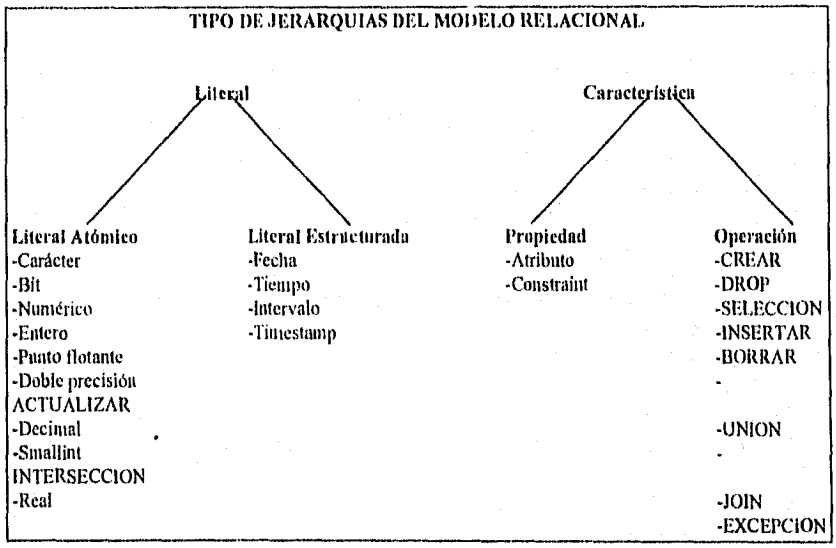
Por ejemplo, utilizando la definición del lenguaje objeto (ODL) de ODMG:

```
interface Student
( extent students)
( attribute String name;
  attribute Short student_id;
   relationship Set -Course- takes inverse Course: :is_taken_by;
)
interface Course ()
) attribute String name;
   relationship Set -Student- is_taken_by inverseStudent: :takes;
)
```

EL MODELO DE OBJETO ODMG ES MAS GENERAL QUE EL MODELO RELACIONAL

El modelo ODMG es un super conjunto de modelos de bases relacionales implementado como un producto DBMS. Los modelos relacionales tipo jerarquías mostrados en la siguiente figura .
Las dos raíces son literales . Las entradas en la columna de tabla pueden ser literales. Una literal puede tener un valor atómico de estructura tipo character o

tipo date. Las propiedades de las características describen tablas y las operaciones que están definidas en ellas .
Una importante diferencia entre el modelo relacional y el modelo de objetos ODMG es que el modelo relacional no soporta usos definidos de subtipos y de tipos de jerarquías. Solo construye tipos que pueden ser usados por aplicaciones. Un programador de objetos debe trasladar objetos dentro de las estructuras predefinidas soportadas por un DBMS relacional a fin de usar el DBMS relacional para almacenamiento persistente. Solamente las operaciones predefinidas de tablas orientadas proporcionadas por SQL pueden ser usadas para accesar a la base de datos relacional. en contraste, el modelo ODMG proporciona una manera para la aplicación semántica para ser expresada en el esquema y soportada directamente por el objeto DBMS.



Las relaciones en el modelo relacional son definidas como restricciones referenciadas de integridad en tablas y no son conceptos de trayectoria transversal.

Han existido varias solicitudes en el modelo relacional que originalmente han concebido la semántica muy poderosa como un modelo objeto. Estas solicitudes están basadas en la igualdad de la noción relacional, en el dominio de la noción

del tipo objeto. Permitiendo que la tabla relacional incluya un conjunto rico de dominios, en lugar de un conjunto relativamente restringido de tipos construidos ofrecidos por relacionales, la habilidad de las bases de datos relacionales para manejar más tipos de datos podría ser ciertamente incrementada. Sin embargo estos dominios necesitan ser capaces de tener aplicaciones extendidas para tipos definidos. La noción de dominio necesita también ser incrementada con la habilidad para especificar operaciones.

3.2. EXTENSION DEL MODELO RELACIONAL PARA QUE SEA ORIENTADO A OBJETOS.

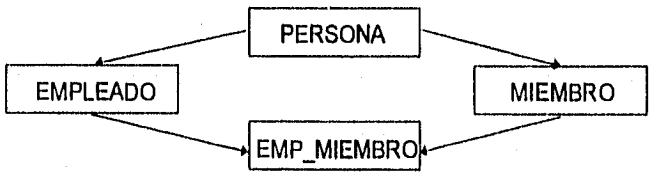
Durante las pasadas tres décadas, la tecnología basada en software para soportar el desarrollo de aplicaciones intensivas de datos sufrió la evolución de cuatro generaciones, comenzando con sistemas de archivos, Sistemas de Bases de datos Jerárquicos, Sistemas de Bases de Datos relacionales. La transición desde la primera generación a la siguiente ha sido motivada por la necesidad de minimizar el costo de desarrollo (que escala rápidamente), así como el costo de mantenimiento y mejoramiento de programas de aplicación.
Los sistemas convencionales (relacionales y prerrelacionales) han servido para mejorar las necesidades de aplicaciones del ambiente para el cuál fueron designadas, es decir , aplicaciones de procesamiento de datos en negocios, tales como control de inventario, nóminas, cuentas por cobrar, etc. Sin embargo, tan pronto como la tecnología de Base de Datos Relacionales abandono los laboratorios de investigación y registro su marca en el mercado, serias limitaciones comenzaron a ser expuestas. En otras palabras, una variedad de aplicaciones comenzó a ser identificada como difícil para implantarse con el uso de Sistemas de Bases de Datos Relacionales. Estas aplicaciones incluyen diseño auxiliado por computadora, ingeniería, ingeniería de software y gestión y administración de procesos (CAD, CAE, CASE Y CAM), sistemas basados en conocimiento (sistemas expertos y "shell" para sistemas expertos), sistemas de multimedia que manejan imágenes, gráficas, voz y documentos textuales; modelos estadísticos y científicos y análisis de programas, sistemas de información geográfica, etc. Estas aplicaciones presentan serias dificultades atribuibles al modelo de datos (Intuitivamente, un modelo de datos es una representación lógica de datos, relacionales e interacción en los datos.)

Las aplicaciones que hemos mencionado requieren, muchas veces:
(a) facilidades para modelar y manejar entidades anidadas complejas (tales como diseño de objetos y documentos compuestos); (b) un conjunto sofisticado

de tipos de datos, por ejemplo, tipos de datos definidos por el usuario, y tipos grandes pero sin estructura (tales como las imágenes, audio y documentos textuales); (c) representación de conceptos semánticos) tales como relaciones de generalización y agregación); (d) el concepto de evolución temporal de datos (por ejemplo, dimensión temporal de datos y mantener versiones de datos), etc.
Estas aplicaciones también presentan importantes dificultades que no están relacionadas con el modelo de datos. Algunas de las aplicaciones requieren de computo altamente intensivo, con un gran volumen de datos en memoria residente, e imponen demandas de ejecución que no pueden ser reunidas por sistemas administradores de datos prerrelacionales. El ambiente de algunas de las aplicaciones también requiere de transacciones de larga duración (por ejemplo, el objeto es tan grande que su actualización toma largo tiempo), transacciones interactivas y cooperativas.
La necesidad para reducir el costo de desarrollo y operación de estas aplicaciones inició la necesidad de un progreso en tecnología de Base de Datos, que esta realizando una transición y además esta impulsando extensiones a sistemas administradores de datos actuales. El núcleo de esta transición radica en el paradigma orientado en los lenguajes de programación orientado a objetos después de la introducción de Simula-67 y Smalltalk-80. Hay dos razones principales por las cuáles la metodología orientada a objetos es un sólido fundamento para la nueva generación de tecnología de Base de Datos.
En primer lugar, la metodología orientada a objetos ofrece un modelo de datos que incluye los modelos de datos de un sistema de Base de Datos convencional. Un modelo de datos orientado a objetos puede representar no solamente los datos, las relaciones y la interacción de datos de modelos de datos convencionales, sino también permite encapsulamiento de datos y programas que operan los datos en un protocolo definido y proporcionan una estructura uniforme para el trato de tipo de datos arbitrarios definidos por el usuario. Algunas relaciones en el modelo de datos, que son difíciles en sistemas de Bases de Datos convencionales, son inherentes en un modelo de datos basados en objetos.
Otra razón es que la metodología orientada a objetos, a través de la noción de encapsulamiento y herencia, esta fundamentalmente diseñada para reducir la dificultad de desarrollo y evolución de sistemas complejos de software. Esta es, precisamente, la meta que motivo a la tecnología de administración de datos a transformar de sistemas de archivos hacia sistemas de Bases de Datos Relacionales. Un modelo de datos orientado a objetos inherente satisface el objetivo de reducir la dificultad de diseñar y desarrollar Bases de Datos complejas, sofisticadas y muy grandes.
Los esfuerzos por desarrollar SMBDOOs pueden mostrarse en dos grandes avenidas. La primera esta fuertemente influenciada por los avances en lenguajes de programación y puede verse como añadir o extender un lenguaje de programación orientado a objetos para convertirlo en un SMBDOO. Un ejemplo de esto es Gemstone, el cual esta montado sobre SmallTalk. La segunda avenida consiste en añadir, extender o modificar los SMBDs existentes para integrarlos

con los conceptos de orientación a objetos. Como ejemplo podemos mencionar a el sistema POSTGRES de la Universidad de California en Berkley.
A pesar de que modificar un SMBD relacional para hacerlo OO es un enfoque poco apreciado en el ámbito de investigación para la obtención de un SMBDOO, dicho enfoque es altamente probable en el ámbito comercial actualmente dominado por modelos relacionales que paulatinamente incorporarán conceptos de OO en nuevas versiones. Posteriormente se ilustrarán dos conceptos de orientación a objetos incorporados a la tecnología de bases de datos relacionales: (1) Jerarquías de herencia y de agregación y (2) Comportamiento.

Modelo de Datos
El corazón de un SMBD es su modelo de datos, el cual es el mecanismo para especificar la estructura de la base de datos y las operaciones permisibles en los datos. Este modelo de datos es una visión abstracta de los datos y logra, con esto, despreocupar al usuario de los detalles de como los datos son físicamente almacenados y administrados. Un modelo de datos permite que la base de datos se vea de una manera intuitiva y acorde con el significado de los datos, como los ve el usuario, ajustándose a varios niveles de abstracción o detalle.
En términos generales, el modelo de datos es una colección de estructuras genéricas y herramientas conceptuales, restricciones y operaciones primitivas para describir datos, para describir relaciones entre los datos, y para asignar una semántica a los datos y condiciones de consistencia de los datos.
La definición de una relación especifica opcionalmente la llave primaria u otras relaciones de las cuales se heredan atributos. La herencia de datos (mas bien atributos) se especifica con la cláusula inherits. Supongamos, por ejemplo, que las personas en la base de datos de un club deportivo son empleados o clientes, y que diferentes atributos definen cada categoría. La relación para cada categoría incluye los atributos de PERSONA y los atributos de la categoría específica. Estas relaciones pueden definirse duplicando los atributos de PERSONA. A continuación se ilustrara la relación y una jerarquía de herencia que puede usarse para compartir la definición de los atributos elementales de una persona.



Las relaciones pueden heredar atributos de más de una superclase. Por ejemplo, EMP_MIEMBRO hereda atributos de las clases MIEMBRO y EMPLEADO. Un conflicto de herencia ocurre cuando el mismo nombre se usa en un atributo para dos clases (por ejemplo, EMP_MIEMBRO hereda Status de MIEMBRO y EMPLEADO). Si los atributos heredados tienen el mismo nombre y tipo, un solo atributo de ese tipo es incluido en la relación que esta siendo definida. De otra manera, la declaración es prohibida.
La herencia proporciona un medio de reducir el esfuerzo necesario para desarrollar y mantener una base de datos. Pero esta capacidad de permitir que una base de datos envejezca graciosamente es bastante más importante. Cuando se utiliza adecuadamente, la herencia permite que se añadan nuevas características y tipos de datos a una base de datos exigiendo solamente unos cambios muy concretos.
Una base de datos relacional tiene únicamente un conjunto limitado de tipos de datos incorporados (ej.: enteros, cadenas, fechas), y un conjunto limitado de operaciones incorporadas (ej.: obtener el valor de un campo, establecer el valor de un campo). Un diseñador de bases de datos relacionales puede crear estructuras de datos mas complejas combinando linealmente tipos básicos, como por ejemplo, campos en registros:

Empleado = Nombre Edad Dirección

Desgraciadamente, como no hay forma de añadir nuevas operaciones para estos nuevos tipos, las operaciones sobre ellos están restringidas a aquellas definidas para los tipos básicos. Una base de datos orientada a objetos proporciona un conjunto equivalente de tipos y operaciones incorporados. Más importante aún: cada objeto de una base de datos orientada a objetos es un miembro de una clase, que determina la estructura del objeto y define que operaciones pueden realizarse sobre él. Las nuevas clases se crean a partir de las clases existentes por medio de la técnica de la subclasificación y obviando los métodos heredados. Esto da como resultado operaciones y tipos de datos complejos arbitrariamente, que pueden ser tratados entonces como si fueran tipos incorporados.
La herencia no solamente ayuda en éste proceso de definir el tipo y las relaciones de los datos a almacenar en una base de datos, sino que también reduce el esfuerzo necesario para acomodar los inevitables cambios en la estructura de la base de datos. La adición de un nuevo tipo de datos es llevada a cabo fácilmente por medio de la subclasificación, con la garantía de que los tipos de datos y métodos existentes no serán afectados por el cambio. La modificación de la representación interna de un tipo de datos o método existente puede estar localizada en aquella clase de jerarquía de clases en la que se define el tipo de datos o método. Finalmente, debido a que los cambios en la estructura jerárquica de la base de datos continúan para asociar los datos que están relacionados lógicamente, el mecanismo de la herencia soporta funciones de base de datos como bloqueo, autorización y consulta.

Tipos de Datos
Las operaciones aritméticas usuales y los operadores usuales de comparación se ofrecen para los tipos de datos atómicos e incluso para cadenas de caracteres. los usuarios pueden definir nuevos tipos de datos mediante la definición de tipos de datos abstractos(TDAs). En realidad todos los tipos de datos del sistema se definen como TDAs. Un TDA se define especificando el nombre del tipo, la longitud de su representación interna en bytes, los procedimientos para convertir a otros tipos y entre representaciones, y un valor por defecto.

Datos de Tipo Procedimiento
Es posible construir arreglos de longitud variable. El segundo tipo de constructor es el tipo procedimiento el cual da la facilidad de almacenar valores de tipo procedimiento dentro de un atributo en un registro. El valor de un atributo de tipo procedimiento es una relación porque eso es lo que resulta del comando retrieve . Más aún, el valor resultante puede consistir de registros de varias relaciones diferentes (es decir, de tipos diferentes) ya que un procedimiento con dos comandos retrieve produce la unión de las tablas resultantes.
Por otro lado, un procedimiento parametrizado puede ser almacenado en un atributo y recibir parámetros cuando es invocado, a continuación ilustraremos con dos ejemplos. Supongamos que los equipos del club deportivo están dados por una relación como la siguiente:

```
create EQUIPOS_CLUB (Deporte = char 25 ,
   capitán = char 25 ....)
```

Los equipos donde participa un miembro de un club pueden ahora representarse por un atributo de tipo procedimiento en la relación MIEMBRO. Este atributo obtiene los equipos en los cuales el miembro está participando. La relación MIEMBRO sería declarada bajo el siguiente esquema:

```
create MIEMBRO (....Equipos = postquel,.....)
```

El tipo de datos postquel indica un atributo de tipo procedimiento. El valor de Equipos es una consulta que recupera las eneadas de la relación EQUIPOS_CLUB que representan las participaciones del miembro en equipos del club.
Por otro lado, los procedimientos parametrizados se utilizan para almacenar en un atributo una consulta que es casi la misma para cada eneada de la relación. Los parámetros de esta consulta son tomados de otros atributos de la relación. Por ejemplo, suponiendo que los miembros del club toman cursos de distintos deportes y que un atributo de la relación miembro representa la lista de clases que el miembro esta cursando. Dada la siguiente relación para las clases de deportes:

```
create CLASES_DEPORTES  (Miembro = char 25 ,
      clase = char 25 )
```

La lista de cursos de Pedro se obtiene por la consulta:

```
retrieve (NombreClase = C.Clase)
  from C in CLASES_DEPORTES
where C. Miembro = "Pedro"
```

Esta consulta serla la misma para cada miembro del club excepto por la constante "Pedro". El procedimiento parametrizado para ésta consulta se define como sigue:

```
define type clases in
   retrieve (Clase_Nombre = C.Clase)
   from C in CLASES_DEPORTES
   where C.Miembro = ·$.Nombre
end
```

El signo $ se refiere a la eneada donde la consulta será almacenada. El parámetro del registro en cada instancia de éste tipo es el atributo Nombre en el registro mismo (el actual). Este tipo de procedimiento se utiliza en la definición de la relación MIEMBRO como sigue:

```
create MIEMBRO (Nombre = char 2 ,....,
    lista_de_Clases = clases
```

De esta manera queda definido un atributo de MIEMBRO que es un procedimiento parametrizado. El atributo puede usarse en una consulta para obtener una lista de las clases que los miembros del club están tomando.:

```
retrieve (S.Nombre, S.Lista_de_Clases.Clase_Nombre)
```

Hemos visto cómo los atributos mismos pueden ser de tipos complejos (procedimiento u arreglo) y cómo todos los tipos de atributos pueden heredarse a una clase. Esto fue una ejemplificación de los conceptos de Agregación y Herencia, respectivamente.
Los Sistemas Administradores de Bases de Datos OO están en proceso de maduración para entrar al mercado comercial y ofrecer no solo las caracterlsticas de todo DBMS, sino todo el potencial del modelo de datos orientado a objetos.
Un Sistema Administrador de Bases de Datos orientadas a objetos (SABDOO) es un Sistema de Bases de Datos (DBMS) que ofrece un modelo de datos orientado a objetos.
Como todo DBMS tradicional, debe proporcionar manejo de discos, administrar datos compartidos, garantizar la integridad de los datos, la seguridad y ofrecer un lenguaje de consulta. Además, para dar soporte a un modelo orientado a objetos, el DBMS debe manejar objetos complejos considerando su identidad, ofreciendo soporte para encapsular datos y comportamiento, estructurándolos en clases y

organizando las clases en jerarquías. ¿Cuáles son las realidades y promesas de esta nueva herramienta para el desarrollo de aplicaciones avanzadas?
Primeramente comenzaremos por presentar las principales características que ofrecen los SABDOO para analizar después sus fortalezas y limitaciones, así como sus tendencias.
Los SABDOO almacenan objetos no solo datos. Objetos que representan, además, entidades u objetos del mundo real que esta siendo modelado en la aplicación. Objetos en el sentido de combinaciones encapsuladas de estructuras de datos (atributos, propiedades) y procedimientos asociados (métodos) que describen su comportamiento. Este mapeo uno a uno reduce la distancia semántica entre el mundo real y el modelo utilizado para representarlo dentro de la computadora. Más aún, asociado a un estilo de programación OO, el SABDOO reduce la diferencia semántica entre el programa y la BD que lo soporta. Comparando con un SABD Relacional, SABDOO ofrece al usuario la posibilidad de manipular objetos con estructuras de datos y operaciones definidas por el mismo, de manera flexible.
Un SABDOO es un Sistema Administrador de Bases de Datos (SABD) que ofrece un modelo de datos orientado a objetos. Las reglas de oro enunciadas a continuación permiten caracterizar un SABDOO; las primeras cinco, por sus funcionalidades en tanto a SABD y las siete restantes por su aplicación del paradigma.

7 reglas para definir un sistema orientado a objetos :

- objetos complejos
- identidad de los objetos
- encapsulamiento
- organización en tipos o clases
- herencia
- sobrecarga y ligado dinámico
- completes computacional

Orientación a Objetos

+

5 reglas para definir un SABD

- persistencia
- administración de disco
- integridad de datos
- concurrencia
- lenguaje de consulta ad - hoc

Características de todo SABD

Características del Modelo Orientado a Objetos
Los aspectos de un SABDOO provenientes del modelo de datos orientado a objetos se comentarán considerando su adaptación a las necesidades planteadas por el enfoque de los SABD. Las características que definen un sistema OO han sido ampliamente comentadas en la descripción de otras areas en donde también

ha tenido gran impacto el paradigma: Análisis y diseño Orientado a Objetos, y Programación Orientada a objetos.

Objetos complejos. Una de las limitaciones del Modelo Relacional, citada con mayor frecuencia, es su descripción de las entidades del mundo real a través de estructuras "planas", con muy pobre riqueza semántica (fidelidad en la descripción de los objetos de la realidad). Los modelos para objetos complejos proponen la posibilidad de construir estructuras de datos que no estén en Primera Forma Normal (Non First Normal Form - NF2), gracias al manejo de atributos con valores no atómicos, sino de tipo tupla, conjunto, lista, o resultado de la aplicación ortogonal de tales constructores.

Identidad de los Objetos. En todo sistema OO, los objetos son reconocidos, diferenciados y referenciados gracias a su identificador (nombre único). Este hecho permite su uso como valores de atributos de otros objetos, ofreciendo gran riqueza en la construcción de estructuras de datos de gran complejidad, que representan objetos del mundo real de manera directa. La noción de identidad ha sido también utilizada como base en la implementación de extensiones para la persistencia de los objetos en algunos lenguajes de programación OO.

Encapsulamiento. El principio de encapsulamiento aplicado a los objetos de una base de datos exige la integración de la estructura y el comportamiento de los mismos en torno de una sola entidad: el objeto. Este principio permite ocultar los detalles de implementación (de los métodos aplicables al objeto) y asegurar la integridad de su parte estructural. Sin embargo, es limitativa desde el punto de vista de base de datos, ya que exige la especificación de métodos especializados a cada tipo de consulta, perdiendo con ello la flexibilidad tradicional ofrecida por un SABD.
Organización de tipos o clases. Durante el proceso de especificación del esquema conceptual de la base de objetos, es natural detectar características o propiedades en común entre varios tipos de objetos. De ahí la ventaja de organizarlos en clases, agrupando bajo la misma clase a todos los objetos que poseen la misma estructura (conjunto de atributos) y que muestran el mismo comportamiento (se aplican sobre ellos los mismos métodos). En el modelo relacional, las entidades que poseen la misma estructura se almacenan como las tuplas de la relación y el modelado de su comportamiento esta en los programas que las manipulan.

Herencia. El mecanismo de la herencia es de gran utilidad en el modelado de una aplicación. De hecho, esta también presente en otros modelos (extensiones al modelo relacional, modelos semánticos), a través de los mecanismos de abstracción de especialización y de generalización, muy frecuentemente usados en el modelado de las entidades de una aplicación. La jerarquía de clases determina una trayectoria de herencia entre la superclase y sus subclases, permitiendo al diseñador reutilizar las definiciones de clases ya existentes,

incorporando sus atributos y métodos. La jerarquía de composición (generada al permitir asignar como valor de un atributo el identificador de otro objeto) es la base de la definición recursiva de un objeto complejo en términos de otros objetos. Un objeto complejo es entonces una instancia de la jerarquía de composición de las clases. En particular, la herencia múltiple plantea problemas especiales que los SABDOO dejan generalmente bajo la responsabilidad del usuario, indicándole únicamente los atributos y métodos en conflicto y que requieren de redefinición.
Sobrecarga y ligado dinámico. La noción de sobrecarga (también denominada polimorfismo) es un concepto en la teoría de tipos en la cual un nombre (o símbolo) puede denotar objetos de varias clases diferentes que se relacionan con una superclase en común. Por lo tanto, los objetos denotados por este nombre pueden responder al conjunto común de operaciones aplicables sobre la clase, pero de manera distinta, según la naturaleza del objeto al cual se aplican. El polimorfismo y la sobrecarga son conceptos que pueden facilitar la tarea del programador en el desarrollo de una aplicación, pero requieren de un lenguaje con ligado dinámico que determine el objeto al cual se aplicará y el código correspondiente, hasta el momento de la ejecución.

Completes computacional. Esta característica exige que todas las operaciones aplicables sobre los objetos no se realicen sino a través del lenguaje explícitamente especificado para ello y no haciendo uso de otras herramientas de menor nivel, para alterar la integridad de la base de objetos.

Características de todo SABD
Se comentara brevemente las características provenientes de los SABD tradicionales que debe ofrecer un SABDOO:
Persistencia. En un lenguaje de programación, todos los objetos manipulados desaparecen al término de la ejecución del programa que los creo y/o manipulo. Por el contrario, en un SABD todos los objetos son creados para persistir y poder ser consultados y/o modificados posteriormente por las aplicaciones. La persistencia es entonces la primera característica que ofrece todo SABD.

Administración del disco. Los avances tecnológicos alcanzados en los dispositivos de almacenamiento permiten almacenar cada vez mayores volúmenes de datos a menor costo. Los SABD son capaces de asegurar el acceso eficiente a los datos almacenados. Sin embargo, las técnicas tradicionales deben adaptarse a las particularidades planteadas por las estructuras de datos del modelo OO garantizando rapidez y flexibilidad en el acceso.

Integridad de los datos. La integridad se define como la propiedad que asegura la calidad de los datos, respetando las propiedades o reglas especificadas en su definición. La integridad global de las bases de datos incluye aspectos como: integridad semántica (fidelidad en la descripción de los objetos del mundo real), seguridad en el acceso (confidencialidad y protección de los derechos de los usuarios) y seguridad en el funcionamiento (recuperación en caso de caída).

Todos estos aspectos que constituyen la integridad de la base de datos, requieren de adaptación al modelo OO, planteando problemas tales como: especificación de reglas sobre los objetos, establecimiento de candados sobre objetos complejos o porciones de ellos, definición de protecciones y autorizaciones de accesos sobre los objetos.
Concurrencia en el acceso. Una de las principales ventajas que ofrece la administración de los datos por un SABD es la posibilidad de poner a disposición de varios usuarios, simultáneamente, la misma fuente de información. Sin embargo, compartir la misma información requiere del establecimiento de protocolos de acceso .
La noción de Transacción que asegura la ejecución de un conjunto de operaciones de manera atómica debe ser adaptada para aplicarse sobre los objetos administrados por el SABDOO, permitiendo la ejecución y control de transacciones "largas".

Lenguaje de Consulta ad-hoc. La principal operación solicitada por los usuarios y sus aplicaciones es la consulta de la información previamente almacenada. Tomando en consideración el gran éxito logrado por lenguajes de consulta como SQL, es evidente la necesidad de continuar ofreciendo un lenguaje de consulta, que tomando en consideración el modelo OO, permita extraer objetos o parte de ellos a través de un lenguaje sencillo, de tipo declarativo.

Fortalezas del Paradigma Orientado a Objetos
Una de las principales fortalezas del paradigma OO aplicado a las bases de datos en su enfoque para el modelado del esquema conceptual, reduciendo la distancia semántica entre los objetos reales, los objetos representados y los objetos almacenados.

El modelo objeto representa de manera mas cercana y fiel el dominio del problema, mapeando las abstracciones de las entidades del mundo real en objetos de la base de datos y facilitando con ello la conceptualización y solución de problemas complejos planteados por las aplicaciones avanzadas. Además, resulta de gran utilidad para el programador contar con un ambiente de desarrollo que permita la representación de entidades del mundo real, como objetos de la base de datos y facilite su manipulación sin necesidad de desmenuzarlos para ser almacenados y reconstruirlos en el momento de ser consultados.
Por otro lado, suprimiendo las funcionalidades (operaciones) de los programas de aplicación e incorporadas en torno de los objetos (a través de los métodos), los SABDOO pueden reducir grandemente la cantidad de código de las aplicaciones. La posibilidad de reusabilidad del código, así como la presencia de buenas librerías de clases, permiten a los desarrolladores de aplicaciones ser mas productivos.
La tecnología de los SABDOO ofrece además los mecanismos para modelar los tipos de datos requeridos por las aplicaciones no tradicionales, para las cuales las funcionalidades provistas por un SABD de tipo Reacional resultan insuficientes.

Desde el punto de vista de la tecnología asociada a los SABDOO, se han alcanzado varios logros:
Se han desarrollado nuevas metodologías para SABDOO, tales como: modelos de datos, lenguajes de consulta, técnicas de indexación, optimización de consultas, modificación del esquema, interfaces de usuario, mecanismos de autorización, métricas de eficiencia, arquitecturas de cliente/servidor y administradores de objetos en memoria.
Se ha realizado experimentación considerable: se han construido prototipos parciales y completos, teniendo como prototipos principales IRIS, ORION. Estos prototipos han sido usados como vehículos de experimentación con el mayor número de características de diseño e implantación especificadas para SABDOO. También han sido utilizados para evaluar la eficiencia y las funcionalidades de la tecnología de SABDOO.

Limitaciones de los SABDOO Actuales y Tendencias
Aunque existen hoy en día prototipos y más aún productos de SABDOO la tecnología es todavía joven y en proceso de maduración. No existe aún gran experiencia acumulada ni el desarrollo de aplicaciones reales, cada etapa es un paso en el aprendizaje, inclusive para los distribuidores de productos. No existen estándares generalmente aceptados y puestos en práctica por los miembros de la comunidad.
Varias iniciativas existen en este sentido y algunos resultados empiezan a alcanzarse.
Otros aspectos sobre los cuales los especialistas de los SABDOO continúan trabajando son:
El factor de escala no ha sido aún probado y no se han realizado pruebas sobre grandes volúmenes de datos. Las características de tipo SABD, como optimización de consultas, control de la concurrencia, seguridad en el acceso y recuperación en caso de caída tienen aún ciertas debilidades. Su eficiencia va mejorando en los nuevos productos.
Un tema de gran debate es la definición e implantación de un lenguaje de consulta orientado a objetos (similar a un ObjectSQL) que establezca la armonía entre el principio de encapsulamiento y la necesidad de conocer el contenido de los objetos por su valor y no por su identificador. Además, por qué limitar al usuario a únicamente consultar el contenido de los objetos a través de procedimientos (métodos) predefinidos.
Nuevas metodologías para análisis y diseño del esquema conceptual de una Base de Datos Orientada a Objetos empiezan a proponerse y a utilizarse para el desarrollo de aplicaciones complejas. Un buen diseño determina la eficiencia del sistema final: De ahí la importancia de acumular experiencia usando las nuevas metodologías.
A pesar de los logros alcanzados en los últimos diez años, quedan aún aspectos por profundizar, declarados como importantes y promisorios, en términos de la relevancia para los investigadores y del desafío técnico que ofrecen. Tales aspectos son: modelado y optimización de consultas, intefaces con el usuario,

metodologías y herramientas para el diseño, mecanismos para manejo de vistas, mediciones de la eficiencia de los sistemas. Además, se señalan otros aspectos no menos relevantes: teoría de tipos y su integración en los lenguajes de bases de datos, características sobre las arquitecturas en la implantación de administradores de objetos para configuraciones de máquinas distribuidas y de cliente/servidor, especificación y reforzamiento de las restricciones de integridad, así como diseño, uso y mantenimiento de bibliotecas reusables de objetos.

4. EJEMPLOS DE BASES DE DATOS ORIENTADAS A OBJETOS EXISTENTES (INVESTIGACION)

4.0 INTRODUCCIÓN

Las bases de datos orientadas a objetos ofrecen parte de la misma funcionalidad que los lenguajes orientados a objetos. Permiten la encapsulación dentro de objetos de los datos y métodos que actúan sobre ellos. Activan métodos mediante mensajes a los objetos. Permiten la declaración de relaciones jerárquicas entre los objetos a través del uso de la herencia. Además, las bases de datos orientadas a objetos ofrecen a las bases de datos tradicionales la funcionalidad de la que carecen los lenguajes orientados a objetos, como persistencia y participación.

4.1 POSTGRES

El modelo de datos THE POSTGRES es un modelo relacional que ha sido extendido con tipos de datos abstractos incluyendo los operadores y procedimientos definidos por el usuario, relación de atributos de tipos de procedimientos, y atributos y herencia de procedimientos. estos mecanismos pueden ser usados para simular una amplia variedad de semántica y construcción de modelos de datos orientados a objetos incluyendo agregación y generalización, complex objetos con subobjetos compartidos, y atributos, de esas referencias tuplas en otras relaciones.
El modelo de datos THE POSTGRES, es una generación extendida del sistema administrador de bases de datos siendo desarrollado en la Universidad de California. El modelo de datos es basado en la idea de extender el modelo relacional desarrollado por Codd con mecanismos generales que pueden ser usados para simular una variedad de construcciones semánticas de modelos de datos. Los mecanismos incluyen :
1) Abstracción de tipos de datos (ADTs),
2) datos de tipo procedimiento y
3) reglas.
Estos mecanismos pueden se usados para soportar objetos complejos o para implementar la herencia de objetos compartidos para un lenguaje de programación orientado a objetos.

Se ha descubierto que algunas descripciones semánticas que no fueron directamente soportadas pueden ser fácilmente adicionadas a el sistema. Consecuentemente, nosotros hemos hecho varios cambios para el modelo de datos y la sintaxis de el lenguaje de consulta que esta documentado aquí.
Esos cambios incluyen soporte para llaves primarias, herencia de datos y procedimientos y atributos de esas referencias de tuplas en otras relaciones.
Lo mas importante de esto es mostrar la herencia que puede ser adicionada para un modelo de datos relacional con solo un numero de cambios para el modelo y la implementación del sistema. Como conclusión podemos decir que el mejor resultado de un modelo de datos orientado a objetos puede ser claramente y eficientemente soportado en un sistema manejador de bases de datos relacional extendido. Las características usadas para soportar esos mecanismos son una abstracción de tipos de datos y atributos de procedimientos.

MODELO DE DATOS

Una base de datos esta compuesta de una colección de relaciones que contiene tuplas las cuales representan el mundo real de las entidades o de relación de ships. Una relación tiene atributos de tipos fijos que representa propiedades de la entidad y relación de ships y una llave primaria. Los tipos de atributos pueden ser atómicos o estructurados. La llave primaria es una secuencia de atributos de la relación, cuando tomen juntos, especialmente identificando cada tupla.
Ejemplificando con una base de datos de una universidad, el siguiente comando define una relación que representa a la gente:

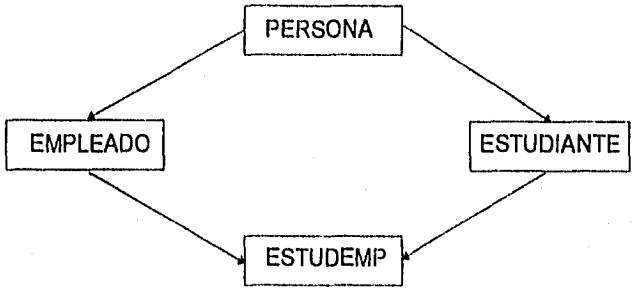
```
create PERSONA( Nombre = char [25],
   Fechnac = date, Estatura = int 4,
   Peso = int4, Dirección = [25],
   Ciudad = char[25], Estado = char [2] )
```

Este comando define una relación y crea una estructura para almacenar las tuplas.
La definición de una relación puede opcionalmente especificar una llave primaria y otras relaciones de las cuales heredan atributos. Una llave primaria es una combinación de atributos que especialmente identifica cada tupla.
Por ejemplo:

```
create PERSONA (....)
key (Nombre)
```

Las tuplas pueden tener un valor para todos los atributos. La especificación de una llave puede opcionalmente incluir el nombre de un operador que es para ser usado cuando la comparación de dos tuplas. Por ejemplo, suponiendo una relación tuvo una llave en la cual el tipo fue definido por el usuario ADT. Si un atributo de tipo box fue parte de una llave primaria, la comparación del operador puede ser especificada desde diferentes operadores box que podrían ser usados

para distinguir las entidades (área equals o box igualmente). El siguiente ejemplo muestra la definición de una relación con una llave de atributo de tipo box este usa el operador área equals (AE) para determinar el valor de la llave igualmente:
create ESTAMPA (Titulo = char[25]), detalle = box)
key (detalle **using** AE)
Los datos de herencia son especificados con la cláusula **inherits**. Suponiendo, por ejemplo, que la gente en la base de datos de la universidad son empleados y/o estudiantes y estos son de diferentes atributos para ser definidos por cada categoría. La relación para cada categoría incluye los atributos de PERSONA y los atributos que deben ser especificados para las categorías. Estas relaciones pueden ser definidas por la respuesta de los atributos de PERSONA en cada definición de relación o por la definición heredada de PERSONA. El siguiente ejemplo muestra las relaciones y la herencia de una manera jerárquica que podría ser usada para compartir la definición de los atributos.



Los siguientes comandos definen otras relaciones que las relaciones de PERSONA definidas arriba:

create EMPLEADO (Depto = char 25,
Nivel = Int2, Coordinador = char 25,
Tittrab = char 25, Salario = Dinero)
inherits (PERSONA)

create ESTUDIA (Fac = char 12,
Nivel = Int2, Profesio = char 20)
inherits (PERSONA)

create ESTUDEMP (EsTrabEstud = bool)
inherits (ESTUDIA, EMPLEADO)

Una relación hereda todos los atributos desde estos padres a menos que un atributo no haga caso en la definición. Por ejemplo, la relación EMPLEADO hereda los atributos de PERSONA que son Nombre, Fechnac, Estatura, Peso,

Dirección, Ciudad y Estado. Las especificaciones de esta llave son también heredadas y nombre es también la llave para EMPLEADO.
Las relaciones pueden heredar atributos de mas de un padre. Por ejemplo, ESTUDEMP hereda los atributos de ESTUDIANTE y EMPLEADO. Un conflicto de una herencia ocurre cuando los nombres de los atributos son iguales en mas de un padre. Por ejemplo ESTUDEMP hereda el nivel de EMPLEADO y ESTUDIA. Si los atributos heredados tienen el mismo tipo, uno de los atributos es incluido en la relación que esta siendo definida. De otra manera, la declaración no es aceptada.
THE POSTGRES es un lenguaje de consulta generalizado de la versión de QUEL, llamado POSTQUEL. QUEL fue extendido en varias direcciones. Primero, POSTQUEL tiene una cláusula **from** para definir variables - tupla mejor dicho que un comando **range.** Segundo, arbitrariamente la expresión relation-valued puede aparecer en cualquier lugar de una relación, este nombre de relación podría aparecer en QUEL. Tercero, transitivamente los comandos clousurey **excute** han sido adicionados para el lenguaje. Finalmente, POSTGRES guarda datos históricos tan rápido que POSTQUEL acepta consultas para ser corridas sobre bases de datos anteriores o sobre cualquier dato que estuvo en la base de datos en cualquier tiempo.
La cláusula **from** fue adicionada para el lenguaje, también la definición de la variable tupla para una consulta podría ser fácilmente determinada en el tiempo de compilación. Esta capacidad fue necesaria para POSTGRES porque el usuario pediría compilar consultas y salvar a estas en el sistema de catálogos. La cláusula **from** es ilustrada en la siguiente consulta de una lista de trabajadores que estudian y quienes son los estudiantes que se encuentran en segundo grado.

retrieve (SE. nombre)
from SE In ESTUDEMP
where SE. EsTrabEstud
  and SE. Nivel = "segundogrado"

La cláusula **from** especifica el conjunto de tuplas sobre las cuales una variable tupla podría encadenar. En este ejemplo, la variable tupla SE encadena sobre el conjunto de empleados que estudian.
Una variable tupla con el nombre igual es definido por cada relación referenciada en la lista destino o la cláusula **where** de una consulta. Por ejemplo, la consulta de arriba podría haber sido escrita:

retrieve (ESTUDEMP, nombre)
where ESTUDEMP, EsTrabEstud
  and ESTUDENP. Nivel = "segundogrado"
Esa noticia del atributo EsTrabEstud es un atributo de valor booleano así que no requiere un valor de pregunta explícita (ESTUDEMP.EsTrabEstud = "true").
El conjunto de tuplas, una variable tupla puede ser cadena de una relación de nombre o una relación de expresión. Por ejemplo, suponiendo si el usuario quiere

obtener todos los estudiantes en la base de datos de quien vive en la ciudad de México; indiferentemente de que ellos sean solamente estudiantes o estudiantes/empleados. Esta consulta puede explicarse de la siguiente manera:

retrive (S.nombre)
from S in Estudia*
where S. ciudad = "Mexico"

El operador "*" especifica la relación formada por tomar la unión de la relación llamada ESTUDIA y todas las relaciones que heredan atributos de esta. Si el operador "*" no fue usado, la consulta obtiene cada tupla en relación con el estudiante (quienes son estudiantes que no sean estudiantes/empleados). En más modelos de datos la herencia soporta la relación nombre a falta de la unión de relaciones sobre la herencia jerárquica (los datos descritos por **ESTUDIA *** arriba). Nosotros escogemos una diferente porque las consultas que involucran uniones serian mas lentas que las consultas sobre una relación simple. Por fuerza el usuario para solicitar la unión explícita con el operador "*", el seria cuidadoso de este costo.
La relación de expresiones puede incluir otro conjunto de operadores, como son: unión (U), intersección ( ), y diferencia (-). Por ejemplo, la siguiente consulta obtiene los nombres de las personas quienes son estudiantes o empleados pero no son estudiantes y empleados:

retrieve (S. nombre)
from S **in** (ESTUDIA **U** EMPLEADO)

Suponiendo una tupla no tiene un atributo referenciado en otra parte en la consulta. Si la referencia esta en la lista destino, la tupla cuando regresa no contendría el atributo. Si la referencia esta en la calificación, la cláusula contiene la calificación que es "falsa".
POSTQUEL también provee un conjunto de operadores de comparación y una relación constructora que puede ser usada para especificar algunas consultas dificultosas de una manera mas fácil que en un lenguaje de consulta convencional. Por ejemplo, suponiendo que pudiera haber varios estudiantes importantes. La representación natural para este dato es para definir una relación separada:

create PRINCIPAL (Snombre = char [25])
    Mnombre = char [25]

en donde Snombre es el nombre del estudiante y Mnombre es el principal. Con esta representación, la siguiente consulta obtiene los nombres de los estudiantes con la misma importancia que Smith:
retrieve (M1.Snombre)
from M1 **in** PRINCIPAL

```
where {(x.nombre) from  x in PRINCIPAL
          where x.Snombre = M1.Snombre}
   C { (x.Mnombre) from x in PRINCIPAL
       where x.Snombre="Smith"}
```

Las expresiones engloban en un conjunto de símbolos ("{...}") que son las relaciones constructoras. La forma general de una relación constructora es:

```
(lista-destino) from from-clausula
        where where-clausula
```

la cual especifica la relación similar como la consulta

```
retrieve (lista-destino)
from from-clausula
where clausula-where
```

Nótese que esa variable tupla definida en el exterior de la consulta (M1 en la consulta de arriba) puede ser usada dentro de una relación constructora pero esa variable tupla definida en la relación constructora no puede ser usada en el exterior de la consulta. La redefinición de una variable tupla en una relación constructora crea una variable distinta como en un bloque de un lenguaje de programación estructurada por ejemplo en PASCAL). Las expresiones de relación valuada pueden ser usados en cualquier lugar en una consulta.
En una base de datos actualizar datos son especificados con comandos de actualización como se muestra en el siguiente ejemplo:

```
/* Adiciona un nuevo empleado para la base de datos. */
append to EMPLEADO (nombre = valor,
edad = valor,....)

/* Cambia el código del estado usando MAP (AntCod, NuevoCod). */
replace P(Estado = MAP.NuevoCod)
from P in PERSONA *
where P.Estado = MAP.AntCod

/* Borra los estudiantes que nacieron antes de este día.
delete ESTUDIA
where ESTUDIA.Nacimiento < "estedia"
```

Las diferentes semánticas de actualización son usadas para todos los comandos de actualización.
POSTQUEL soporta los comandos de clausura transitiva desarrollada en QUEL
El comando "*" continua ejecutando hasta que las tuplas no son obtenidas o actualizadas (**append***, **delete***, or **replace***). Por ejemplo. la siguiente consulta

crea una relación que contiene quienes son todos los empleados que trabajan para Smith:

```
retrieve* into SUBORD(E. Nombre, E.Coordinador)
from E in EMPLEADO, S in SUBORD
where E.Nombre = "Smith"
or E.Coordinador = "S.Nombre"
```

Este comando continua para ejecutar el comando **retrieve-into** hasta que no haya cambios hechos en la relación SUBOR.
Ultimamente, POSTGRES salva datos borrados o modificados en una relación así que las consultas pueden ser exceptuadas en datos históricos. Por ejemplo, en la siguiente consulta se va a obtener quienes son los estudiantes que vivían en México el 1o. de Agosto de 1980:

```
retrieve* (S.Nombre)
from S in  ESTUDIA ("Agosto 1o. 1980")
where S.Ciudad = "Mexico"
```

Los datos especificados en los brackets siguiendo el nombre de la relación especifica la relación en el tiempo designado. Los datos pueden ser especificados en muchos formatos diferentes y opcionalmente pueden incluir en que tiempo y en que día. La consulta de arriba solo examina quienes son los estudiantes que no son estudiantes y empleados. Para buscar el conjunto de todos los estudiantes, con la cláusula **from** seria:

```
from S in ESTUDIA * ("Agosto 1o. 1980")
```

Las consultas pueden también ser ejecutadas en todos los datos que están ocurriendo en la relación o que estuvieron en algún tiempo en el pasado. La siguiente consulta obtiene quienes son todos los estudiantes que han vivido siempre en México:

```
retrieve (S.Nombre)
from S in ESTUDIA ()
where S.Ciudad = "México"
```

La notación "( )" puede ser adicionada para cualquier nombre de relación.
Las consultas pueden también ser especificadas en datos que estuvieron en la relación durante un periodo en un tiempo dado. El periodo de tiempo es especificado dando un principio y un tiempo final, como se muestra en la siguiente consulta que obtiene quienes son los estudiantes que vivieron en México en el mes de Agosto de 1980:

```
retrieve (S.Nombre)
from  S  in ESTUDIA*("Agosto 1, 1980",
```

"Agosto 31, 1980")
where S.Ciudad = "México"

POSTGRES también soporta versiones de relaciones. Una versión de relación puede ser creada desde una relación o una versión instantánea. Una versión es creada por especificación de la relación base como se muestra en el comando

create version MYGENTE **from** PERSONA

esto crea una versión nombrada MYGENTE, derivada desde la relación PERSONA. Los datos pueden ser obtenidos y actualizados en una versión justamente como una relación. Actualizar datos en la versión no se modifica en la relación base. Sin embargo, la actualización de datos para la relación base son propagados para la versión a no ser que el valor haya sido modificado. Por ejemplo, Si su cumpleaños de Jorge es cambiado en MYGENTE, el comando **replace** cambia su cumpleaños en PERSONA que podrá no estar propagado para MYGENTE.
Si un usuario no quiere actualizar en la relación base para propagar en la versión, el puede crear una versión instantánea. Una versión instantánea es una copia de la ocurrencia contenida de una relación. Una versión instantánea es creada por el siguiente comando:

create version TUGENTE
from PERSONA ("ahora")

La versión instantánea puede ser actualizada directamente por consecuencia de los comandos de actualización en la versión. Pero, los datos de actualización para la relación base no son propagados para la versión.
Una combinación de comandos proporciona una combinación de cambios hechos para una versión posterior dentro de la relación base. Un ejemplo de este comando es:
merge TUGENTE **into** PERSONA

esto podría combinar los cambios hechos anteriormente dentro de PERSONA. La combinación de comandos utilizándolos en un procedimiento semiautomático para resolver la actualización de datos para la relación fundamental y el conflicto de la versión.

TIPOS DE DATOS.

POSTGRES proporciona una colección de tipos y estructuras atómicas. La predefinición de tipos atómicos incluyen: int2, int4, float4, float8, bool, char y date. La aritmética estándar y la comparación de operadores son proporcionados para los tipos de datos numéricos y tipo fecha (date) y las cadenas estándar y comparación de operadores para los arreglos de caracteres. Los usuarios pueden

extender el sistema por adición de nuevos tipos atómicos usando una abstracción de tipo de datos (ADT) facilita la definición.
Todos los tipos de datos atómicos son definidos para el sistema como ADT's. Un ADT es definido por especificación del nombre del tipo, la longitud de la representación interna en bytes, procedimientos para convertir de una representación externa a una interna por un valor y una representación interna a una externa, y por ausencia de un valor. El comando

```
define type int4 is (longint = 4,
    InputProc = CarToInt4,
    OutputProc = Int4ToCar, Default = "0")
```

determina el tipo int4 el cual es predefinido en el sistema. CharToInt4 y Int4ToChar son procedimientos que son codificados en un lenguaje de programación convencional.
Los operadores en ADT's son definidos por especificación del número y tipo de operandos, el tipo de regreso, precedencia y asociatividad del operador, y el procedimiento que implemente a esto. Por ejemplo, el comando

```
define operator "+"(int4, int4) returns int4
    is (Proc = Plus, Precedencia = 5,
        Asociatividad = "Izq"
```

determina el operador plus. La precedencia es especificada por un número. Los números largos implican mas grande la precedencia. Los operadores predefinidos tienen las precedencias que se muestran a continuación

Precedencia	**Operadores**
80	
70	not - 8(unary)
60	*/
50	+ - (binary)
40	< < > >
30	= =/=
20	and
10	or

Esas precedencias pueden ser modificadas por cambios en la definición de operadores. La asociatividad puede ser izquierda o derecha dependiendo en la semántica deseada. Este ejemplo define un operador denotado por un simbolo (" +"). Los operadores pueden también ser denotados por identificadores como se muestran en la tabla de arriba.
Otro ejemplo de la definición de un ADT es el siguiente comando que define un ADT este representa cajas.

```
define type caja is (LongInt = 16,
    InputProc = CarToCaja,
    OutputProc = CajaToCar, Default = " ")
```

La representación externa de una caja es una cadena de caracteres que contiene dos puntos, estos representan la parte superior izquierda y la parte inferior derecha de la caja. Con esta representación, la constante

"20,50:10,70"

describe una caja en la cual la esquina superior izquierda es (20,50) y la esquina inferior derecha es (10,70), CarToCaja toma una cadena de caracteres y regresa una representación de una caja de 16 bytes. CajaToCar es la inversa de CarToCaja.
La comparación de operadores pueden ser definidos en ADT's estos pueden ser usados en métodos de acceso o en optimización de consultas. Por ejemplo, la definición

```
define operator AE (caja, caja) return bool
    is (Proc = CajaAE, Precedencia = 3,
        Asociatividad = "Izq" , Sort  = CajaArea,
        Hashes, Restrict = AERSelect,
         Join  =  AEJSelect, Negator = CajaAreaNE)
```

determina un operador "areaequals" en cajas, En adición a la información de la semántica acerca del mismo operador, esta especificación incluye información utilizada por POSTGRES para construir indexacion y para optimizar consultas usando el operador. Por ejemplo, suponiendo la relación de una figura fue definida por

```
create FIGURA(Titulo = char ( ), Item = caja)
```

y la consulta

```
retrieve (FIGURA,all)
where FIGURA, Item AE "50,100:100,50"
```

fue ejecutado. La propiedad de sortear del operador especifico el procedimiento para ser usado para sortear la relación; si una mezcla de sort unida a la estrategia fue seleccionada para implementar la consulta. Este también especifica el procedimiento para usarse cuando se construye una indexacion ordenada en un atributo de tipo caja. La propiedad Hashes indica que este operador puede ser usado para construir una indexacion hash en un atributo caja. Nótese que cualquier tipo de indexacion puede ser usada para optimizar la consulta que se encuentra arriba. Las propiedades de Restrict y Join especifican el procedimiento

que es para ser llamado por la optimización de la consulta para computo restrict y el join son selectivos respectivamente de una cláusula que complica el operador. Estas propiedades selectivamente especifican procedimientos, estos regresaran a valores de punto flotante entre 0.0 y 1.0 esto indica los atributos selectivamente que da el operador. Ultimamente la propiedad Negator especifica el procedimiento para ser usado para comparar dos valores en donde el predicado de una consulta requiere del operador para ser negado como en:

retrieve (figura, **all**)
where not (figura, item
AE "50,100:100,50)

El comando **define operator** también puede especificar un procedimiento, este puede ser usado si el predicado de la consulta incluye un operador que no sea conmutativo. Por ejemplo, el procedimiento del conmutador para "el área menor que" (ALT) es el procedimiento que implementa "área mayor que o igual) (AGE). Mas detalles en el uso de estas propiedades son dadas en cualquier otra parte.
El tipo de constructores son proporcionados para definir tipos de estructuras (arreglos y procedimientos) estos pueden ser usados para representar datos complejos. Un arreglo es un tipo de constructor que puede ser usado para definir una variable o un tamaño fijo de arreglo. Un tamaño fijo de arreglo es declarado por especificación de tipos de elementos y un limite máximo del arreglo, el cual define un arreglo de 25 caracteres. Los elementos del arreglo son referenciados por indexacion; el atributo por un entero entre 1 y 25.
En el arreglo el tamaño de una variable es especificado por omisión el limite máximo en el tipo de constructor. Por ejemplo, en un arreglo de caracteres el tamaño de una variable es especificado por "char ." En los arreglos el tamaño de una variable son referenciados por indexacion, el atributo por un entero entre 1 y el limite superior actual del arreglo. La función predefinida regresa el tamaño actual del limite superior. POSTGRES no impone un limite en la longitud del tamaño de una variable del arreglo.
La construcción en funciones son proporcionadas para añadir los arreglos y para traer una parte de ellos. Por ejemplo, dos caracteres en un arreglo pueden ser adicionados usando la concatenación del operador ("+") y una parte de un arreglo, dividiendo 2 caracteres de una parte de 15 en un atributo llamado x, que puede ser traído por la expresión "x 2:15."
El segundo tipo de constructor permite valores del tipo de procedimiento para ser almacenado en un atributo. Los valores de un procedimiento son representados por una secuencia de los comandos de POSTQUEL. El valor de un atributo del tipo de procedimiento es una relación porque es una recuperación del comando returns. Además, el valor puede incluir tuplas de diferentes relaciones (de diferentes tipos) porque un procedimiento compuesto de dos comandos recuperados regresa la unión de ambos comandos. Nosotros llamamos a una relación con diferentes tipos de tuplas una multirelación. La interface del lenguaje de programación POSTGRES proporciona un mecanismo cursor-like, llamado un

portal, para traer valores de las multirelaciones. Sin embargo, ellos no son almacenados por el sistema (solamente las relaciones son almacenadas).
El sistema proporciona dos clases de tipos de constructores de procedimiento: variable y parámetros. Una variable en un tipo de procedimiento permite un diferente procedimiento de POSTQUEL para ser almacenado en cada tupla mientras el tipo de procedimiento parametrizado almacena el procedimiento similar en cada tupla pero con diferentes parámetros. A continuación se ilustrara el uso de una variable del tipo de procedimiento mostrado por otro camino para representar el principal estudiante. Suponiendo la relación de un DEPARTAMENTO fue definida con el siguiente comando:

```
create DEPARTAMENTO (Nombre = char 25,
```

El mejor estudiante puede entonces ser representado por un procedimiento en la relación ESTUDIANTE que recupera la tupla apropiada de DEPARTAMENTO. El principal atributo podria ser declarado como sigue:

```
create STUDENT (....Principal = postquel,....)
```

Los tipos de datos de postquel representan un tipo de procedimiento. El valor en Principal podria ser una consulta que traiga la relación de las tuplas del departamento que representa el estudiante mas bajo. El siguiente comando adiciona un estudiante a la base de datos que es el principal en materias, en matemáticas y en ciencias computacionales:

```
append STUDENT (Nombre = "smith",....
    Principal =
    "retrieve (D.all )
    from D in DEPARTAMENTO
    where D.Nombre = "Math"
        or D. Nombre = "CS"")
```

Una consulta que referencia el atributo Principal regresa el string que contiene el comando POSTQUEL. Sin embargo, dos notaciones son proporcionadas para ser ejecutada la consulta y regresa el resultado antes que la definición. Primero, la notación nested-dot implica ejecutar la consulta como se ilustra.

```
retrieve (S.Nombre, S. Principal.Nombre)
from S in STUDENT
```

el cual imprime una lista de nombres de los principales estudiantes. En el resultado de la consulta en Principal esta implícita la unión con la tupla especificada por el resto de la lista destino. En otras palabras si un estudiante es el principal en dos materias, esta consulta regresaria dos tuplas con el nombre del atributo repetido. La unión implícita es ejecutada para garantizar que una relación es regresada.

El segundo camino para ejecutar la consulta es para usar el comando de **execute.** Por ejemplo, la consulta

```
execute (S.Principal)
from S in ESTUDIANTE
where S.Nombre = "Smith"
```

regresa una relación que contiene las tuplas de DEPARTAMENTO para todos los principales de Smith.
Los tipos de procedimiento parametrizado son usados cuando la consulta para ser almacenada en un atributo es cercanamente la misma para todas las tuplas. El parámetro de la consulta puede ser tomado para otros atributos en la tupla o ellos pueden ser explícitamente especificados. Por ejemplo, suponiendo un atributo in ESTUDIANTE esta para representar a los estudiantes actuales de la lista de clases. Se obtiene la siguiente definición para inscripciones:

```
create INSCRIPCION (Estudiante = char 25,
    Class = char 25 )
```

La lista de las clases de Bill puede ser recuperada por la consulta

```
retrieve (ClaseNombre = E.Clase)
from E in INSCRIPCIONES
where E.Estudiante = "Bill"
```

Esta consulta puede ser la misma para todos los estudiantes excepto para el constante que especifica el nombre del estudiante:
Un tipo de procedimiento parametrizado podría ser definido para representar esta consulta como sigue:

```
define type clases is
    retrieve (ClaseNombre = E. Clase)
    from E in INSCRIPCIONES
    where E.Estudiante = S.Nombre
end
```

El símbolo del signo de dollar ("$") referencia a la tupla en el cual la consulta es almacenada (la tupla actual). El parámetro para cada instancia de este tipo (una consulta) es el Nombre del atributo en la tupla en la cual la instancia es almacenada. Este tipo es entonces usado en el comando **create** como sigue

```
create ESTUDIANTE (Nombre = char 25,....
    ClaseLista = clases)
```

para definir un atributo que representa la lista de clases de los estudiantes actuales. Este atributo puede ser usado en una consulta para regresar una lista de estudiantes y de clases que ellos están tomando.

```
retrieve (S.Nombre, S.ClaseLista.Clasenombre)
```

Esta noticia para una tupla de un estudiante particular, la expresión "$.Nombre" en la consulta se refiere a el nombre del estudiante. El símbolo "$" puede ser el concepto como de una tupla de variable limitada para la tupla actual.
Los tipos de procedimientos parametrizados son extremadamente tipos utilizados, pero algunas veces esto es inconveniente para almacenar los parámetros explícitamente como atributos en la relación. Consecuentemente, una notación es proporcionada para permitir los parámetros para ser almacenados en el valor del tipo de procedimiento. Este mecanismo puede ser usado para simular tipos de atributo que referencian tuplas en otras relaciones. Por ejemplo, suponiendo que se quiere un tipo que referencia una tupla en la relación DEPARTAMENTO. Este tipo puede ser definido de la siguiente manera:

```
define type DEPARTAMENTO (int4) is
   retrieve  DEPARTAMENTO, all)
   where DEPARTAMENTO, oid = $1
end
```

El nombre de la relación puede ser usado para el tipo nombre porque las relaciones, tipos y procedimientos han separado el nombre por espacios. La consulta en el tipo DEPARTAMENTO recuperara una tupla del departamento especifico dando un identificador especial al objeto (oid) de la tupla. Cada relación tiene implícitamente definido un atributo nombrado oid que contiene el identificador especial. El atributo oid puede ser accesado pero no actualizado por el usuario que consulta. Los valores oid son creados y mantenidos por el sistema de almacenamiento POSTGRES. El argumento formal para este tipo de procedimiento es el tipo de un objeto identificador. El parámetro es referenciado dentro de la definición por "$n" donde n es el numero del parámetro.

Un argumento actual es suministrado donde un valor es asignado para un atributo de tipo DEPARTAMENTO. Por ejemplo, una relación CURSO puede ser definida para representar información acerca de un curso especifico incluyendo el departamento que ofrece este. El comando **create** es:

```
create CURSO (Titulo = char 25 ,
   Dept = DEPARTAMENTO,...)
```

El atributo Dept representa el departamento que ofrece el curso. La siguiente consulta adiciona un curso a la base de datos:

```
append CURSO(
```

```
    Titulo = "Introducción de Programación",
     Dept = DEPARTAMENTO (D,oid))
from D in DEPARTAMENTO
where D.Nombre = "ciencia de la computación"
```

El procedimiento DEPARTAMENTO llamado en la lista destino es implícitamente definido por el comando " **define type** ". Este valor del constructor del tipo especificado da argumentos actuales que son tipos compatibles con los argumentos formales, en este caso un int4.
Los tipos de procedimientos parametrizados que representan referencias para tuplas en una relación especifica son también comúnmente utilizados para el plan de proporcionar soporte automático para estas. Primero, todas las relaciones creadas podrán hacer un tipo que representa una referencia para una tupla definida implícitamente similar a la definida arriba. Y segundo, esto seria posible para asignar una tupla-variable directamente a una tupla consultando el atributo. En otras palabras, el asignamiento para el atributo Dept es escrito en la consulta de arriba como.

```
...     Dept = DEPARTAMENTO (D.oid)....
```
puede ser escrito como
```
...     Dept = D ...
```

Los tipos de procedimiento pueden también ser usados para implementar un tipo que referencie una tupla en una relación arbitraria. La definición de tipo es:

```
define type tupla (char , int4) is
    retrieve ($1,all)
    where $1.oid = $2
end
```

El primer argumento es el nombre de la relación y el segundo argumento es el oid de la tupla deseada en la relación. En efecto, este tipo define una referencia para una tupla arbitraria en la base de datos.
El tipo de procedimiento tupla puede ser usado para crear una relación que representa a la gente quien ayuda con found raising

```
create VOLUNTARIO(Persona = tupla,
  TiempoDisponible = Integer, .... )
```

Porque los voluntarios pueden ser estudiantes, empleados, o gente quienes ni son estudiantes ni son empleados, los atributos de Persona pueden contener una referencia para una tupla en una relación arbitraria. El siguiente comando adiciona todos los estudiantes para VOLUNTARIO:

```
append VOLUNTARIO(
```

```
        Persona = tupla (relación (S), S.oid))
    from S in ESTUDIANTE *
```

La función relación predefinida regresa el nombre de la relación para la cual la variable tupla S es limitada.
El tipo tupla será también caso especial para hacer esto mas conveniente. Tupla será un tipo predefinido y esto será posible para asignar variables tupla directamente para atributos del tipo. Consecuentemente, el asignamiento para Persona escrito arriba como

```
... Persona = tupla (relacion (S), S.oid) ...
```

puede ser escrito

```
... Persona = S ...
```

PROCEDIMIENTOS DEFINIDOS POR EL USUARIO

Los procedimientos definidos por el usuario son escritos en un lenguaje de programación convencional y son usados para implementar operadores ADT o para mover una computación de un proceso de aplicación front-end para el back-end del proceso DBMS.
Moviendo una computación para el back-end abre posibilidades para el DBMS para precomputar una consulta que incluye la computación. Por ejemplo, suponiendo que una aplicación front-end necesita para traer la definición de una forma desde una base de datos y para construir una estructura de datos en memoria principal que el tiempo de corrida del sistema de forma usado para desplegar la forma en la pantalla de la terminal para los datos de entrada o despliegue. El diseño de una relación convencional de base de datos seria almacenar los componentes de la forma (los títulos, y definición de los campos para los diferentes tipos de estos tal como campos escalares, campos tabla, campos gráficos) en muchas relaciones diferentes.
Un ejemplo del diseño de una base de datos es:

```
create FORMA(FormaNombre,....)

create CAMPOS (FormaNombre, CampoNombre,
   Origen, Altura, Amplio,
   CampoClase, ...)
create CAMPOESCALAR(FormaNombre,
   CampoNombre, TipoDato,
    FormatoDisplay,...)


create CAMPOTABLA(FormaNombre,
```

```
    CampoNombre, NumdeRen,...)

  create TABLACOLUMNS(FormaNombre,
    CampoNombre, ColumNom, Altura,
    Amplio, CampoClase, ...)
```

La consulta que trae la forma desde la base de datos puede ejecutarse en lo mas mínimo de una consulta por tabla y sorteando a través del regreso de tuplas para construir la estructura de datos en memoria principal. Esta operación puede tomar menos que dos segundos para una aplicación interactiva. El convencional Sistema Manejador de Bases de Datos relacional no puede satisfacer este tiempo reservado.
El acercamiento para resolver estos problemas es para mover la computación, esta construcción de la estructura de datos en memoria principal para el proceso de base de datos. Suponiendo el procedimiento HacerForma construye la estructura de datos dando el nombre de una forma. Usando el mecanismo del tipo de procedimiento parametrizado definido anteriormente en donde un atributo puede ser adicionado a la relación FORMA que almacena la representación forma calculada por este procedimiento. Los comandos

```
  definine type formarep is
    retrieve (rep = HacerForma($.FormaNombre))
  end
  adicatributo (FormaNombre, ...,
    FormaDatoEstruc = formarep)
  to FORMA
```

define el tipo de procedimiento y adiciona un atributo a la relación FORMA.

La ventaja de esta representación es que POSTGRES puede precalcular la respuesta para un atributo de un tipo de procedimiento y almacenar este en la tupla. Por precalculación de la representación de la estructura de datos en memoria principal, la forma puede ser traída de la base de datos por recuperación de una tupla simple:

```
  retrieve (x = FORMA, FormaDatoEstruc)
  where FORMA., FormaNombre = "foo"
```

El tiempo real obliga a traer y desplegar una forma que puede fácilmente encontrar si todo el programa puede estar haciendo una recuperación de una tupla singular para traer la estructura de datos y llamar el procedimiento de librería para desplegar este. Este ejemplo ilustra la ventaja de calcular el movimiento (construyendo una estructura de datos en memoria principal) desde el proceso de aplicación del Sistema Manejador de Bases de Datos.

Un procedimiento es definido para el sistema por especificación de nombres y tipos de los argumentos, el regreso del tipo, este lenguaje es escrito en donde la fuente y el código objeto es almacenado. Por ejemplo en la definición:

```
define procedure EdadEnAños(date) returns int4
  is (lenguaje = "C", nombrearchivo = "EdadEnAños")
```

define un procedimiento EdadEnAños que toma el valor date y regresa la edad de la persona. El argumento y el regreso de type son especificados usando los tipos POSTGRES . En donde el procedimiento es llamado, esto es permitido en la representación interna de POSTGRES para el tipo. Nuestro plan para permitir procedimientos para ser escritos en varios y diferentes lenguajes incluyendo C y Lisp, los cuales son dos los dos lenguajes que han sido usados para implementar el sistema.
POSTGRES almacena la información acerca de un un procedimiento en el sistema de catálogos y dinámicamente carga el código del objeto cuando este es llamado en una consulta. La siguiente consulta usa el procedimiento EdadEnAños para recuperar los nombres y las edades de todas las personas en el ejemplo de la base de datos:

```
retrieve (P.Nombre,
  Edad = EdadEnAños(P.Cumpleaños))
from P in PERSONA"
```

Los procedimientos definidos por el usuario pueden también tomar un argumento de una variable tupla. Por ejemplo, el siguiente comando define un procedimiento, llamado Comp, este toma una tupla EMPLEADO y calcula la compensación de las personas acordando para algunas la formula que involucra varios atributos en la tupla (el empleado, su trabajo, el puesto que ocupa, y el salario.):

```
define procedure Comp (EMPLEADO)
 returns int4 is (lenguaje = "C" ,
  NombreArchivo = "Comp1")
```

Esta llamada de tipo de procedimiento parametrizado es definido para cada relación automática así que el tipo EMPLEADO representa una referencia para una tupla en la relación EMPLEADO. Este procedimiento es llamado en la siguiente consulta:

```
retrieve (E.Nombre, Compensacion = Comp (E))
  from E in EMPLEADO
```

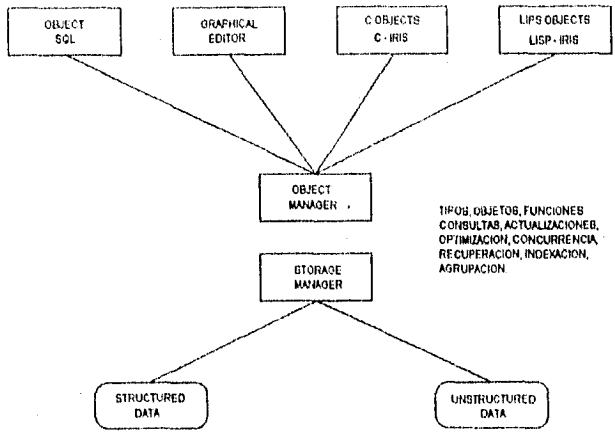
La función C implementa este procedimiento que esta permitiendo una estructura de datos que contiene los nombres, tipos, y valores de los atributos en la tupla.

En los procedimientos definidos por el usuario pueden estar pasando tuplas en otras relaciones que heredan los atributos en la relación declarada como el argumento para el procedimiento.

4.2 IRIS

El manejador de bases de datos Iris es una investigación prototipo de una generación siguiente al DBMS siendo desarrollada en los laboratorios de Hewlett-Packard alrededor de 1989. Se han explorado las características y funcionalidades de esta nueva base de datos a través de una serie de más y más capacidad de prototipos. El prototipo de Iris está implementado en C sobre estaciones UNIX HP-9000/350.

Iris esta deseado para conocer las necesidades de nuevas aplicaciones en las bases de datos, tales como información de oficinas y sistemas de bases de conocimientos, análisis de ingeniería, dimensiones y diseño de hardware y software. Estas aplicaciones requieren un rico conjunto de capacidades que no son soportadas por la generación actual del DBMS's. En adición para el requerimiento usual para la permanencia de datos, bajar y recuperar información, las nuevas capacidades son necesidades que incluyen la construcción de modelos de datos, bases de datos directos que soportan por inferencia, tipos de datos novel (gráfica imágenes, voz, texto, vectores, matrices), interacciones largas con lapsos en minutos en la base de datos para muchos días, y múltiples versiones de datos. Los datos compartidos pueden ser condicionados a nivel objeto en ambos sentidos y al mismo tiempo compartidos consecutivamente, permitiendo dar un objeto para ser accesado por aplicaciones que pueden ser escritas en diferentes lenguajes de programación orientados a objetos.
El sistema manejador de bases de datos "The Iris" esta siendo diseñado para encontrar estas necesidades:



La figura anterior muestra las capas que componen la arquitectura de Iris. El centro de esta arquitectura es el Administrador de Implementaciones de Objetos del Modelo de Datos Iris el cual cae dentro de la categoria general de los modelos orientados a objetos que soportan un alto nivel de abstracción estructural tal como clasificación, generalización / especialización y agregación tan bien como el comportamiento de abstracciones. El traductor del procesador de consultas a las consultas Iris y funciones dentro de un formato de álgebra relacional, esto es optimizado y luego interpretado otra vez en la base de datos almacenada. Mejor dicho que inventando totalmente un nuevo formalismo sobre la base para nuestro sistema de comportamiento correcto, nosotros confiamos en el álgebra relacional como en nuestra historia de computación. Esto tiene demostración muy convenientemente en términos de coexistencia y de migración, existiendo aplicaciones en la base de datos.

El Manejador de Almacenamiento Iris es actualmente un subsistema de almacenamiento relacional convencional. Esto suministra accesos asociativos y capacidad para actualizar una relación simple a un tiempo, incluyendo soporte de transacción.

Como más de otros sistemas de bases de datos, Iris es una accesible vía de una sola posición de Interfaces interactivas o a través de módulos de interfaces empotradas en lenguajes de programación. Los módulos de interface, tales como esas etiquetas C-Iris y Lips-Iris, facilita accesos para la persistencia de objetos de varios lenguajes de programación. La construcción de interfaces es hecha posible por un conjunto de subrutinas definidas en lenguaje C, claro que si la interfaz es del manejador de objetos.

Al mismo tiempo, tres interfaces interactivas son soportadas. Una es simplemente un manejador para la interface Administrador de Objetos. Otra interface

interactiva, es SQL Objeto (OSQL) es una extensión de SQL de objetos orientados. Se ha escogido extender SQL en lugar de inventar totalmente otro lenguaje por la importancia de SQL en la comunidad de base de datos y porque se ha explorado la posibilidad de extensiones parecidas. Una tercera fase interactiva, es el editor gráfico, que permite al usuario explorar interactivamente el tipo de estructura Iris Metadata tan bien como la estructura Interobject Relation Ship definida sobre una base de datos Iris data. Esto es descrito en Object-C y soporta la actualización para esquemas y datos.
Nosotros estamos también explorando tres clases de interfaces programáticas. La primer clase es un empotramiento sencillo de OSQL dentro de varios lenguajes. La segunda clase es una encapsulación del Sistema Manejador de Bases de Datos Iris como un lenguaje de programación de objetos a quienes corresponden métodos para las funciones en la interface de subrutina C para el Administrador de Objetos Iris. La tercer clase de interface programática se está explorando, ésta después de un largo término de investigación dentro de objetos persistentes, se intenta crear un lenguaje de programación de objetos transparentemente persistente a través de las aplicaciones y lenguajes.

MANEJADOR DE OBJETOS IRIS

El Manejador de Objetos Iris implementa el modelo de datos por proporcionar soporte para la definición de esquemas, manipulación de datos y procesamiento de consulta. El modelo de datos, el cual esta basado en las tres construcciones, objetos, tipos y funciones que soportan herencia y propiedades genéricas, obligaciones, complejidades o datos no normalizados, funciones definidas por el usuario, control de versiones, inferencia y tipos de datos extendidos. Las rutas del modelo pueden ser encontradas en trabajos previos sobre DAPLEX, el Modelo de Datos Integrados, la extensión DAPLEX y el lenguaje TAXIS.

OBJETOS

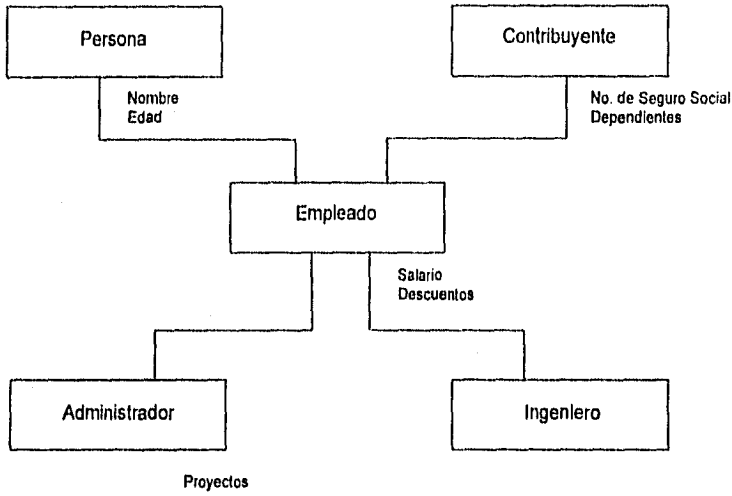
Los objetos representan entidades y conceptos de dominios de aplicaciones que han sido modelados. Ellos son entidades especiales en la base de datos con sus propias identidades y existencias, y ellos pueden ser referidos para ser indiferentes de sus valores de los atributos. Por ejemplo, cada objeto tiene asignado un amplio sistema, especialmente para identificar objetos, OID. Este soporte referencial es integramente y es una mayor ventaja sobre los modelos de datos de registros orientados en los cuales los objetos son representados como registros, pueden ser referidos para solamente en términos del valor de su atributo.
Los objetos son descritos por su comportamiento, y pueden sólo ser accesados y manipulados por el significado de sus funciones. Tan grande como la semántica de las funciones similares que quedan, la base de datos puede ser físicamente tan bien como lógicamente reorganizada fuera de la afectación de la aplicación de

programas. Esto suministra una gran cantidad de abstracciones de datos y de independencia de los datos.
Los objetos son clasificados dependiendo de su "tipo". Los objetos que pertenecen de un tipo similar comparten funciones comunes. Los tipos son organizados dentro de un tipo jerárquico con funciones heredadas. Consecuentemente un objeto puede tener tipos múltiples. Los objetos sirven como argumentos para funciones y pueden ser aplicados como resultado de funciones. Una función puede ser aplicada para un objeto solamente si la función es definida en un tipo para el cual el objeto pertenece.
Por una propiedad de un objeto nosotros pretendemos que una función de nuestro modelo es propiedad de los objetos de IRIS con funciones. Las funciones pueden ser definidas incluyendo predicados y funciones de múltiples argumentos, proporcionando directamente soporte binario o n-ary relacionships.
El modelo de datos IRIS se distingue entre objetos literales, tal como cadenas de caracteres y números, y objetos no literales tal como personas y departamentos. Los objetos no literales son representados internamente en la base de datos por identificadores de objetos. Los objetos literales no tienen accesibles el uso de identificadores de objetos y son directamente representables. Tal como ellos no pueden ser creados, destruidos o actualizados por los usuarios.
El manejador de objetos proporciona explícitamente sencillez creando y borrando objetos no literales, y para asignación y actualización de valores para sus funciones. La referencia íntegramente es soportada: cuando un objeto dado es borrado, todas las referencias para el objeto son borradas también.

TIPOS Y JERARQUÍAS DE TIPOS

Los tipos son nombrados colecciones de objetos. Los objetos que pertenecen a un tipo igual que comparten funciones comunes. Por ejemplo, todos los objetos que pertenecen al tipo Persona tienen una función Nombre y una función Edad. Las funciones son calculadas, definidas en tipos. Ellas son aplicables a las instancias de los tipos. En efecto, por lo tanto, los tipos son obligados, esto es, un tipo obligado permite a funciones que puedan ser aplicadas para un objeto de este tipo.
Los tipos son organizados en un tipo de estructura que soporta generalización y especialización. Un tipo puede ser declarado para ser un subtipo de otro tipo. En este caso, todas las instancias de el supertipo. Esto significa que estas funciones son definidas en el supertipo y son también definidas en el subtipo. Nosotros decimos que estas funciones son heredadas por el subtipo. En IRIS, un subtipo hereda todas las funciones definidas en este supertipo. Esto es diferente de otros sistemas de objetos en los cuales la función de un supertipo puede opcionalmente, ser heredado por un subtipo. Por supuesto, un supertipo IRIS puede tener instancias estas no pertenecen para cualquiera de estos subtipos.
El tipo de estructura de Iris es directamente una gráfica aciclica. Un tipo dado puede tener subtipos múltiples y supertipos múltiples.

El siguiente esquema ilustra un tipo de gráfica con cinco tipos, cada tipo tiene un número de propiedades. El tipo EMPLEADO es directamente un subtipo de los tipos PERSONA y CONTRIBUYENTE, y el tipo EMPLEADO, tiene este mismo dos subtipos directamente, Administrador e ingeniero.
Las instancias del tipo EMPLEADO también pertenecen a los tipos PERSONA y CONTRIBUYENTE. Las funciones definidas en PERSONA y en CONTRIBUYENTE son heredadas por EMPLEADO. De este modo, los objetos de EMPLEADO tienen todas las seis funciones: Salario, Descuentos, Nombre, Edad; Numero de Afiliación al Seguro Social y Dependencia. Nosotros podemos crear un nuevo tipo, ASESOR , como sigue. Todas las funciones no actualmente pertenecen a tipos.



Los nombres de las funciones pueden ser sobrecargadas, esto es las funciones definidas en diferentes tipos pueden tener nombres idénticos aunque sus definiciones pueden diferir. Así que el diseño de una base de datos puede introducir una función en la forma mas general; por definición esto es un tipo general y una educación tardía de la definición de función para los mas especializados subtipos. Por ejemplo el tipo EMPLEADO puede tener una función general salario, visto que los tipos Ingeniero y Administrador tienen las funciones salario que son específicas para las dos categorías de trabajo. Esta aproximación para el diseño es llamado stepwise refinamiento por especialización.
En donde una sobrecarga de funciones es aplicada para unos objetos dados, una simple función especificada puede ser seleccionada en el tiempo de aplicación. La función especificada es determinada por el objeto para el cual es aplicado. Si el objeto pertenece a varios tipos los cuales todos tienen funciones específicas de un dado nombre, la función del tipo mas específico es seleccionado. Si un solo tipo

mas especifico no puede ser encontrado, el usuario especifica reglas para seleccionar la función que aplicara. Estas reglas son especificadas por familias de funciones que compartirán el mismo nombre. Estos conceptos también se aplican para funciones que toman varios argumentos.
El tipo de Objeto es el supertipo de todos los otros tipos y por lo tanto contiene muchos objetos. Los tipos son objetos, ellos mismos y su relación de ships para subtipos, supertipos e instancias son expresadas como funciones en el sistema.
En otro soporte gracioso para evolución de bases de datos, el manejador de objetos permite el tipo de gráfica para ser cambiada dinámicamente. Por ejemplo, nuevos tipos pueden ser creados, existiendo tipos borrados y los objetos pueden ganar o perder tipos por todas partes de su tiempo de vida. Por ejemplo, existiendo un objeto asesor puede favorecer a un empleado.
Sin embargo, nuevos subtipos/supertipos existiendo tipos entre relación de ships actualmente no pueden ser creados. Así que, nosotros no podemos crear el tipo asesor como un subtipo del tipo empleado.
Tales operaciones podrán ser soportadas en el futuro.
Por lo tanto Un tipo dado puede tener múltiples subtipos y múltiples supertipos (herencia múltiple).
Ejemplo de creación de tipo:

Create type Consultor **Subtype of** Persona, Pagador; (El tipo Consultor hereda de los tipos Persona y Pagador)

FUNCIONES Y REGLAS

En Iris una función es un calculo que puede o no puede regresar un resultado. Las funciones son definidas en tipos y son aplicables para las instancias de los tipos. Iris distingue funciones de la actualización de estas. Así que una función recuperada puede no tener defectos. Iris soporta las funciones definidas por el usuario y estas son compiladas, almacenadas y ejecutadas bajo el control del sistema manejador de la base de datos. La especificación de una función Iris, lógicamente consiste de dos partes: una declaración y una implementación. Podemos notar que SQL también permite ambas especificaciones para ser cambiadas dentro de una simple declaración **create** function.
DECLARACIÓN DE FUNCIONES

Una declaración de función especifica el nombre de la función y el número y tipos de estos parámetros y resultados. Por ejemplo:

Create Function Matrimonio (Persona p) ----> <Persona cónyuge, Boda, Fecha> **Forward;**
Crea una función llamada Matrimonio. La cláusula **Forward** declara que la implementación de la función será especificada más tarde. Como una conveniencia, si forward es omitido, de cualquier manera la función da una implementación almacenada.

Una función puede regresar un resultado compuesto como se muestra en el siguiente ejemplo, en donde el resultado de la función contiene ambos cónyuges y la fecha del matrimonio. Dando una implementación, la función matrimonio puede ser de la siguiente manera (asumiendo que "bob" es una variable limitada para un objeto Persona):

```
Select  s, d for each Persona S, Boda d
  where <s,d> = Matrimonio (bob);
```

Este enunciado puede ser abreviado como:

```
Select each Persona S, Boda d
  where <s, d > = Matrimonio (bob)
```

o más simplificado:

```
Select Matrimonio (bob);
```

La declaración de la función es también usada para especificar la participación que enclerra el número de ocurrencias de cada parámetro y valor del resultado. Por ejemplo, la interface tiene mecanismos para declaración de funciones que son resultados requeridos o especiales. Un resultado requerido pretende que el valor del resultado pueda existir para cada valor del parámetro posible en la base de datos. Un resultado especial pretende valores distintos de parámetros que serán diferentes valores de resultados. Nosotros también distinguimos entre funciones con valores simples y funciones con valores múltiples.
Las funciones pueden retornar un solo valor o retornar múltiples valores (multivaluadas). Por ejemplo, una función que retorne todos los hijos de una persona, es una función multivaluada.
La información sobre los objetos es modelada en Iris usando *funciones predicado*. El concepto atributo es modelado usando funciones cuyos valores son derivados de predicados. Por ejemplo, un predicado Persona_Edad conecta a las personas con sus edades, este puede definirse como:

Create function Persona_Edad(Persona p, Integer a) **forward** ;

donde Persona_Edad(john,31) es verdadero si la persona tiene la edad especificada. Una vez implementada la función Persona_Edad se pueden derivar las funciones:

Edad(Persona) = Integer ;
Persona_with_Edad(Integer) = Persona ;
donde son inversas una de otra. Muchas funciones simples son fácilmente invertibles mientras que sean derivadas de predicados.

Separar la declaración de una función de su implementación soporta la abstracción de datos, permitiendo a los usuarios cambiar la implementación dinámicamente sin afectar los programas de aplicación.

Tipos de funciones

Funciones almacenadas . Una forma de implementar una función es almacenar esta como una tabla, mapeando los valores de entrada con sus correspondientes valores resultantes. Como una tabla, puede ser implementada y accesada usando técnicas standard de bases de datos relaciónales. La operación **cluster** permite al usuario especificar que una función será implementada en esta forma. Por ejemplo:

Create Function Matrimonio(Persona p) -> <Persona cónyuge, CharString fecha> **forward** ;
Cluster Matrimonio ;

Este comando causa la creación de una tabla con tres columnas : persona, cónyuge y fecha.

Funciones derivadas . La definición de una función puede ser especificada en términos de otras funciones.
En general, la definición de funciones puede contener consultas e incluir llamadas a cualquier función implementada. Por ejemplo:

Create function Emp_Manager (Empleado e) -> Manager as
select m **for each** Manager m
where m = Department_Manager (Emp_Department (e))

Funciones foráneas . Son funciones implementadas en algún lenguaje de programación de propósito general como C. Aunque Iris no puede optimizar la implementación de estas funciones, sí puede optimizar su uso. La invocación de funciones foráneas es facilitada por la interfaz entre el sistema de procesamiento de las consultas y el subsistema de almacenamiento.
Ejemplo:

Create function FechaCompara (CharString fecha1, CharString fecha2)
-> Integer as **link** 'fecha_comp' ;

Reglas. En Iris las reglas son simplemente modeladas como funciones. A través de estas se maneja la inferencia. Por ejemplo, dada una función *Parent*, se puede definir una función Abuelo como sigue:

Create function Abuelo (Persona p) -> Persona as
Select gp **for each** Persona gp

where gp = Padre (Padre (p)) ;

El prototipo actual de Iris soporta solo conjunción, disyunción y reglas no recursivas, la negación y recursión están siendo estudiadas.

Operaciones de actualización (Set, Add, Remove)

Una operación de actualización cambia el comportamiento futuro de funciones almacenadas o derivadas, por ejemplo:

Set Department_Manager (sales_dept) = john ;

Esta operación causará invocaciones futuras a la función con el parámetro sales_dept para retornar el objeto actual ligado a la variable john.

Add. Se usa para adicionar valores.

Ejemplo:

Add Miembro (ventas_depto) = p **for each** Persona p
where p = Miembro (Depto_juguete) ;

(Causa que todos los miembros del Departamento de Jugueteria también se conviertan en miembros del Departamento de Ventas.)

Remove. Elimina.

Ejemplo:

Remove Miembro (ventas_depto) = e **for each** Empleado e
where e = Miembro (ventas_depto) **and** Edad(e) > 70 ;

(Elimina del Departamento de Ventas todos los empleados mayores de 70 años)
Select. Operación de consulta

Delete. Esta operación puede borrar cualquier objeto, tipo o función definido por el usuario.

Ejemplo:

Delete type Ingeniero ;

(Causa que el tipo Ingeniero sea borrado. Por demás, todas las funciones con argumento o resultado Engineer serán borradas. No obstante, las instancias de este tipo no son borradas, solo pierden el tipo.

Control de versiones

Existen aplicaciones que desean preservar estados alternativos para una entidad particular, para ello necesitan la existencia de un mecanismo de versiones de objetos que proporcione el control de acceso a esos valores. Este mecanismo ha sido implementado como parte integral del Manejador de Objetos. Iris separa objetos correspondientes a cada versión.
El control de versiones en Iris proporciona una forma de direccionamiento indirecto, donde los objetos pueden hacer referencias genéricas a otros objetos.
Iris también brinda la facilidad de crear versiones de aquellos que fueron objetos no versionados originalmente, debido a que no siempre es posible conocer de antemano cuáles partes de un diseño se colocan bajo una versión de control.

Ejemplo:

create version from mod1 **instance** mod1v1 ;

En este ejemplo se convierte un objeto no versionado a versionado. El resultado es un nuevo objeto "mod1v1" el cual será creado con el mismo tipo de usuario que el original, pero adquirirá el tipo del sistema Version y será la primera versión del conjunto de versiones.. El objeto original "mod1" perderá su tipo del sistema Unversioned y adquirirá el tipo del sistema Generic, mantendrá su OID original y actuará como instancia genérica del nuevo conjunto de versión creado.
Los comandos checkin, checkout, lock y unlock permiten la creación y manipulación de versiones y el control compartido de versiones entre usuarios.
Los objetos pueden ser bloqueados con checkout y desbloqueados con checkin.

Ejemplo:

checkout mod1v1 **key** modkey1 **as** mod1v2 ;

(Crea una nueva versión "mod1v2" que puede ser modificada . La llave de bloqueo es retornada en la variable "modkey1")

Procesamiento de consultas

Las consultas en Iris son expresadas en términos de funciones y objetos. El Manejador de Almacenamiento se ocupa del álgebra relacional y de las tablas. El procesamiento de las consultas es compartido por dos módulos: el traductor de las consultas y el intérprete.
El traductor compila las consultas desde su representación de objetos a una representación del álgebra relacional. El intérprete evalúa las consultas

transformadas invocando al Manejador de Almacenamiento para accesar a la base de datos y funciones foráneas para accesar a otras fuentes de datos.
Sobre las interfaces de Iris, veamos un poco más a fondo el **Object SQL**
Existen tres extensiones principales hechas al SQL para adaptar este al modelo de objetos y funciones, estas son:

- Los usuarios manipulan tipos y funciones, más que tablas.
- Los objetos pueden ser directamente referenciados, más que indirectamente, a través de sus llaves.
- Las funciones definidas por el usuario y las del sistema Iris pueden aparecer en las cláusulas where y select.

Veamos algunos ejemplos:

Create type Persona
(nombre Charstring **required**,
dirección Charstring,
telefono Charstring) ;

Create type Aprobado **subtype of** Persona
(experiencia en diversos Tópicos) ;

Create type Autor **subtype of** Persona ;

(Creación de instancias de un tipo):

Create Aprobado (nombre, experiencia) **instances**
Smith ('Albert Smith', software),
Jones ('Isaac Jones', (finanzas, ventas)) ;

(Adición de objetos a un tipo)

Add type Autor **to**
Jones,
Robinson ;

4.3 VBASE

VBase es un ambiente de desarrollo orientado a objetos que combina un lenguaje procedural de objetos y la persistencia de objetos en un sistema integrado. Fue elaborado por la Compañía Ontologic alrededor de 1989 y está implementado para máquinas Sun (ambiente UNIX) y VAX/VMS. Actualmente la compañía se llama Ontos y el sistema también se denomina Ontos, se encuentra en la versión 3.0, está escrito en C++, soporta una interfaz SQL orientada a objetos y está disponible para la mayoría de las plataformas UNIX.
Como aspectos del lenguaje de VBase este incluye: una fuerte tipificación, parametrización, capacidad de tener tipos miembros de objetos agregados, entre otros. Soporta las relaciones entre objetos de uno-a-uno, uno-a-muchos y muchos-a-muchos, el mecanismo inverso y métodos "triggers" (disparadores).
VBase está basado en el paradigma de tipos abstractos de datos.
En VBase el comportamiento de un objeto está dado por la combinación de propiedades y operaciones. Las propiedades representan el comportamiento estático y las operaciones el comportamiento dinámico.

Arquitectura del Sistema

Tiene tres capas o niveles:

- *Nivel de abstracción.* Implementa el meta-modelo de objetos, proporcionando soporte para la herencia, el despacho de operaciones, la combinación de métodos, etc. . Es el nivel donde más interactúan los clientes.

- *Nivel de representación.* Es donde se hacen las referencias semánticas y se traduce la notación simbólica del nivel de abstracción en objetos denotables.

- *Nivel de almacenamiento.* Es el responsable del "mapeo" del nivel de representación en almacenamiento físico.

Cada nivel de VBase es implementado dentro de Vbase mismo. Cada nivel tiene una especificación y una implementación.
Para soportar la persistencia de objetos el sistema permite:

- Manipular gran número de objetos y consecuentemente manipular un gran espacio de almacenamiento para dichos objetos.

- Compartir objetos de datos entre múltiples procesos/usuarios, lo que implica coordinación (control de concurrencia) para evitar la corrupción semántica de la base de datos de objetos.

- Estabilidad del software tal que el espacio del objeto es mantenido en un estado consistente en la fase del sistema o errores medios (transacciones).

VBase ofrece la funcionalidad del método de ligadura dinámica basado en la jerarquía de tipos, como hacen todos los sistemas de objetos. También es totalmente polimórfico.

Componentes del sistema

Antes de la versión actual presentaba dos interfaces del lenguaje: TDL (para la definición de tipos del lenguaje) y COP (C Object Processor). TDL es usado para especificar tipos de datos abstractos, o sea, es usado para definir tipos de datos y especificar sus propiedades y operaciones asociadas. COP es usado para escribir el código para implementar las operaciones y para escribir programas de aplicación.
El sistema incluye también un conjunto de herramientas para ayudar al desarrollo como: editor, un "object browser", un "debugger" , una interfaz SQL con extensiones de objetos y un programa verificador que checa consistencia de imágenes compiladas contra una base de datos de objetos.
TDL es un lenguaje propietario. Es estructurado con características en común con lenguajes como Pascal, Modula y Algol. Los tipos son las entidades más comunes definidas por TDL. Un tipo sirve como un nexo para el comportamiento de sus instancias. Este determina las propiedades y define las operaciones de las instancias.
Esta versión del sistema permite sólo la especificación de un supertipo, dicho supertipo coloca la definición de tipo en la jerarquía de tipo. El comportamiento es heredado vía la jerarquía de tipo de la manera esperada.
COP es un superconjunto estricto del lenguaje C. Cualquier programa que compila con C standard compilará con COP Está actualmente implementado como un preprocesador el cual emite código C estándar.

Mecanismos básicos de abstracción

Existe una taxonomía de tipos con subtipos que heredan propiedades y operaciones de su supertipo. Los subtipos pueden adicionar comportamiento más específico agregando propiedades y operaciones adicionales y pueden redefinir comportamiento ya existente. Cuando una operación es invocada esta es despachada de acuerdo al tipo de objeto de la invocación.

a-Tipificación fuerte

Después que los tipos de datos son definidos por TDL, es escrito el código en COP y compilado contra la base de datos de objetos. El compilador de COP es una aplicación de bases de datos y usa el tipo de información de la base de datos para el chequeo de tipo que sea posible en tiempo de compilación. Cuando el chequeo estático de tipos no es posible, es diferido a tiempo de ejecución.
Las operaciones definidas en los tipos son implementadas por métodos escritos en COP.

Note que todos los objetos variables son declarados con la palabra adicional obj. Cuando no es posible determinar la compatibilidad de tipos en tiempo de compilación, el programador usa la palabra *assert* . Con ella se difiere el chequeo de tipos hasta tiempo de ejecución.

b- Relaciones inversas

Cuando una propiedad es declarada como inversa de otra en cualquier momento que se modifique una, se modificará la otra.

c- Combinación de métodos

La combinación de métodos resulta cuando un método redefinido invoca su redefinición. VBase usa '$$' para este propósito.
Cada definición de operación puede incluir una cláusula *method* y cláusulas *triggers* , o sea, cada operación está potencialmente asociada con un método llamado método base y con un número arbitrario de métodos triggers. La secuencia de ejecución comienza con el primer trigger en la cláusula triggers. '$$' transfiere la ejecución al próximo fragmento de código.

Funcionalmente '$$' se comporta como una llamada a una función.

d- Triggers

Los triggers pueden conectarse tanto a propiedades como a operaciones para generar cualquier comportamiento deseado.
En VBase los triggers son muy usados para aumentar la creación y borrado de métodos.

e- Excepciones

En VBase se incluye un mecanismo para manejo de excepciones específico. Cuando se encuentra una operación *raise* o la palabra *except* significa que hay un manejo de excepción.

Veamos un **ejemplo**
(Código TDL para dos definiciones de tipos)

```
define type Part
   supertypes = {Entity} ;
   properties = {
     partID: Identifier;
     name: optional String ;
     components: distributed Set[Part]
         inverse componentOf ;
```

```
      componentOf: Part inverse components ;
    } ;

    operations = {
      ...
      ...
      ...
      connect (p: Part, to: Part,
        keywords
          optional using: Connector )
        raises (BadConnect)
        method (Part_Connect)
        returns (Part) ;

      refines delete (p: Part)
        triggers (Part_deleteTrigger) ;
    } ;

    PRIVATE
      properties = {
        displayImage: optional Image ;
        isRootComponent: Boolean := False ;
      } ;

end Part ;

define Type Pipe
  supertypes = {Part} ;

  properties = {
    length: Integer := 0 ;
    diameter: Integer := 0 ;
    leftConnection: Part ;
    rightConnection: Part ;
    threadtype: optional ThreadType := Threadtype$ScrewThread ;
    isInsulated: optional Boolean := False ;
  } ;

  operations = {
    refines connect (p: Pipe, to: Part,
      keywords
        optional usingConnector: Connector ;
      raises (ThreadtypeMisMatch, IncompatibleMaterials)
      method (Pipe_Connect)
      returns (Part) ;
```

```
        ...
        ...
        ...
        refines delete (p: Pipe)
          raises (CannotDelete)
          triggers (Pipe_deleteTrigger) ;
    } ;

end Pipe ;

define Type ThreadType is enum (ScrewThread, PolThread) ;

(Código en COP para el método Pipe_Connect)

method
obj Part
Pipe_Connect (apipe, toPart, usingConnector)
obj Pipe aPipe ;
obj Part toPart ;
keyword obj Connector usingConnector;
{
  obj Part connectedPart ;
  obj Pipe toPipe ;
  int j ;
  if (hasvalue (usingConnector))
  {
   connectedPart = Pipe$Connect(aPipe, usingConnector) ;
   return (Part$Connect (connectedPart, toPart, using: usingConnector)) ;
  }
  toPipe = assert (toPart, obj Pipe) ;

  except (ia: IllegalAssert)
  {
      PipeSystem$ErrorPrint (aPipe, toPart, "can only connect Pipes to others
Pipes") ;
  }

  if (aPipe.threadtype != toPipe.threadtype)
   raise (ThreadTypeMismatch) ;
  if ( ! Pipe$MaterialsCompatible (aPipe, toPipe))
   raise (IncompatibleMaterials) ;
  aPipe.leftConnection = toPipe ;
  toPipe.rightConnection = aPipe ;
  connectedPart = $$ (aPipe, aPart) ;
  return (connectedPart) ;
```

```
}
```

Características avanzadas del sistema

a- Parametrización (Tipos genéricos)

VBase tiene la capacidad de especificar el tipo de los objetos contenidos dentro de objetos agregados.

Ejemplo:

```
{
  obj Array[Animal] MyZoo ;
  obj Fruit akiwi ;
  akiwi = myZoo[3] ;
}
```

b- Definición de constantes y variables en TDL

Estas variables pueden ser referenciadas luego directamente en COP

Ejemplo

```
define Constant myDefault := Null ;
define Constant No := False ;
define Variable PartSerialNumber: Integer := 0 ;
```

Luego en COP pudiera hacerse:

```
nextNum = $PartSerialNumber ++ ;
```

c- Enumeración, Unión

TDL permite la definición de tipos enumerados y de uniones, por ejemplo:

```
define type Day is enum (Monday, Tuesday, ...) ;
define type BlockScope is union (Type, Module, Environment, Directory,...);
```

d- Agrupamiento

El sistema posee la capacidad de agrupar objetos en disco y en memoria. Esto es muy usado, por ejemplo en aplicaciones de ingeniería para agrupar objetos que son parte-de o componentes-de.

e- Operaciones libres

Son operaciones que no están asociadas con ningún tipo, o sea, procedimientos simples libres de asociación con algún tipo.

Procesamiento de transacciones y control de concurrencia.
VBase implementa el mecanismo de transacciones por medio de la creación de un área de trabajo separada para cada transacción. Todo el trabajo es hecho copiando objetos en el área de trabajo y operando sobre ellos allí; no se hacen cambios directamente en la base de datos.
Si una transacción se ejecuta satisfactoriamente, las referencias a aquellos objetos modificados por la transacción son adicionadas a una lista guardada por VBase en un área de trabajo especial llamada write set. El tiempo en el cual las referencias son adicionadas también es almacenado. Esta información es usada por el algoritmo de interferencia para determinar si había existido una colisión entre transacciones concurrentes.
El control de concurrencia del VBase asegura que las transacciones no sean abortadas debido a la interferencia vía protocolo de bloqueo (lock), en otras palabras, VBase proporciona un tipo llamado Lock con un protocolo muy simple como la vía más fácil para hacer un arbitraje cuando dos o más transacciones deciden accesar al mismo objeto.

Representación y almacenamiento en VBase.
VBase tiene una arquitectura abierta. Una arquitectura abierta es aquella en la cual los niveles de abstracción, representación y almacenamiento del sistema son distintos, explícitos, ortogonales y accesibles al programador.
Hacer los niveles explícitos es un requerimiento obvio, hacerlos distintos y ortogonales es lo que realmente define la apertura del sistema. Esto permite al programador accesar a cualquier nivel del sistema hasta abajo, hasta el nivel de bits y bytes. Si los niveles son ortogonales, el programador sólo tiene que cambiar lo que es realmente necesario. Cualquier nivel puede ser cambiado sin afectar los otros niveles. Por ejemplo, uno puede diseñar un nuevo manejador de almacenamiento para el sistema de bases de datos el cual corra un algoritmo de compresión sobre cadenas sin cambiar la interfaz del lenguaje abstracto. En un sistema de bases de batos orientadas a objetos, como el VBase, esto permite al desarrollador no sólo adicionar nuevos tipos de datos abstractos al sistema, sino también proporciona nuevas implementaciones para esos tipos (si es deseado).

En adición a las propiedades definidas por los objetos de Tipo Abstracto, cada objeto VBase hereda del tipo "Entity" propiedades referenciandas al Tipo

Abstracto, al Manejador de Representación y al Manejador de Almacenamiento, cada uno de los cuales es otro objeto.
El sistema VBase define los tipos abstractos RepManager y StorageManager que definen el protocolo para el manejo de la representación y el almacenamiento. Creando nuevos subtipos de estos tipos del sistema, los desarrolladores pueden crear nuevos manejadores de almacenamiento y representación. Estos pueden ser usados entonces para instanciar objetos sin alterar el tipo abstracto.
VBase hace uso de esta capacidad de subtipos para definir muchos manejadores de representación diferentes cuando el implícito no es apropiado. También hace uso de manejadores de almacenamiento alternos para diferenciar entre procesos temporales, locales, objetos y persistencia de objetos.

4.4 SIM

Sim es una demostración de base de datos de técnica disponible basado en un modelo de datos semánticos similar al Hammer y McLeod's SDM. Sim tiene dos efectivos primarios de modelamiento. El primero es estrechar la puerta entre la percepción del usuario del mundo real de datos y el modelo conceptual impuesto por la Base de Datos debido al nombramiento de tres suposiciones o limitaciones. El segundo objetivo es permitir tanto como sea posible que los datos semánticos puedan ser definidos en un esquema y hacer que el sistema de base de datos sea responsable para mantener una integridad . SIM provee un conjunto rico de construcciones para la definición de los temas , incluyendo aquellos para la generalización especificada, jerarquías en modelamiento por gráficas directas aciclica, relaciones interobjeto y reestructuras de integridad. También trabaja novedosamente y fácil de usar un modelo de manipulación de datos en lenguaje inglés para recuperación y actualización de operaciones. Este trabajo describe la clave de las características de los modelos de SIM, la arquitectura del sistema y sus consideraciones de implementación.

INTRODUCCION

Un modelo de datos consiste de reglas para definir la estructura lógica de datos asociada con ooperaciones. El poder expresivo, la simplicidad y la libertad de la implementación de los detalles son algunas características deseables del modelo de datos. Los modelos relacionales fueron los primeros en enfatizar tanto la estructura cómo los objetos de manipulación de los modelos asi como la independencia de almacenamiento. Es construido en Fundamentos Matemáticos y tiene algunas ventajas como son la simplicidad y sus conceptos completos. Sin embargo, su principal debilidad del modelo relacional es la falta de una

poderosa expresión semántica y requiere que los conceptos de una aplicación sean fragmentadas para ubicar el modelo forzando el esquema resultante y las consultas de la Base de Datos de manera que se pierde su naturaleza conceptual. Los pasos artificiales en las formulaciones de consultas introducen nivel de indirección y tienen sobre tonos de procedimiento.
El SIM (Manejador de Información semántica) fue iniciado en 1982 en Unisys con el objetivo de producir la próxima generación de sistemas administradores de bases de datos (DBMS). El modelo semántico fue escogido porque es rico, expresivo conceptualmente natural y provee una trayectoria de crecimiento para los usuarios del DM SII. DMSII, es un manejador de bases de datos basado en las estructuras de trabajo de los modelos de datos que corren con las máquinas Unisys Serie A. Han sido instalados en bases para usuarios y han sido usados para inplementar grandes aplicaciones complejas. SIM inicialmente ha sido construido sobre la base del DMSII y descansa en DMAII para transacción, y manejo del cursor Y/O. Sin embargo arquitectura del sistema es virtualmente diseñada de manera que cualquier fuente de datos, incluyendo otros sistemas de bases de datos, puede ser sustituida en el lugar del DMSII. SIM forma la base para el Infoexec, el cuál provee un arreglo de bases de datos y herramientas de aplicación, incluyendo ADDS.

EL MODELO SEMANTICO DE DATOS

Las entidades, relacionan Ships entre entidades, mecanismos de abstracción y las obligaciones de integridad son generalmente reconocidas como hechos importantes de los modelos semánticos. Una entidad es un objeto abstracto que corresponde al mundo real o conceptual en un ambiente de aplicación. Una colección semánticamente hablando orma una clase. Las entidades no existen aisladas, ellas están relacionadas unas con otras en varias maneras, y la noción de atribución describe la relación de Ships entre entidades. Un atributo de una entidad define como es relacionado con otras entidades o quizá de la misma clase o de otra. Las entidades son representadas por identificadores de sistemas definidos y su existencia no depende de cualquiera de sus atributos. Los atributos de una entidad está dicho que son disponibles si su rango es de un número especial, en un sistema de clases definidos (por ejemplo, la clase de todas las cadenas). Los atributos pueden ser evaluados de una manera simple o multivariable. La generalización permite que cada miembro de la clase pueda ser relacionada a un miembro de una ó más clases genéricas llamadas superclases .
La noción de generalización puede ser aplicada exitosamente, brindando una jerarquía de clases. Los tipos de datos, opciones atribuidas y los predicados de aserción son las principales técnicas para la restricción de especificación en modelos semánticos de datos. Los modelos semánticos proveen fuertes hechos ó ventajas que pueden ser usados de una manera natural para restringir los valores de un atributo. Las escrituras fuertes también permiten a los usuarios hacer asociaciones sin significados entre componentes de datos. Las opciones de atributo como especiales distinguen entre máximos y mínimos que son utilizadas

para especificar la estructura integral de esos datos. Estas opciones son suficientes para describir los usuales uno a uno, uno a muchos y muchos a muchos en relaciones similares. La aserción predicada especifica condiciones que deben ser satisfechas por entidades de una clase. Ellas son establecidas como predicación que mantienen una base de datos en todos los tiempos ó bien en predicados probados después de las acciones DML que son ejecutados. Las aserciones basadas en transacciones también son permitidas.

DEFINICION DE UN ESQUEMA EN SIM

La mayoría del trabajo en Modelo Semántico de datos ha sido concentrado en su utilidad como herramientas de diseño de bases de datos. Mientras que algunos prototipos de Bases de datos pueden implementar un conjunto seleccionado de propiedades de modelos semánticos existentes, hacia el mejor de nuestros conocimientos, SIM es uno de los mas grandes, completos en modelos comercialmente implantados .

ENTIDADES

Las entidades son objetos abstractos que pueden representar una idea ó una cosa en un mundo real. Ellos juegan un papel en el SIM a las grabaciones en el DBMS convencional ó tuplas en el sistema relacional. Sin embargo, las entidades son construidas lógicamente y no implican particularmente implementaciones físicas como grabaciones y tuplas que así lo hace. Las entidades y objetos en sistemas orientados a objetos son similares en su estructura pero diferentes en que los objetos pueden poseer descripciones algorítmicas llamadas métodos que las entidades de SIM no tienen.

CLASES

La unidad primaria de una encapsulación de datos en SIM es una clase, la cuál representa una significante colección de entidades. Una clase es también una base de clases o subclades . Una base de clases es definida independientemente de todas las otras clases en la base de datos, mientras que la subclase es definida en una ó más clases, llamada superclases.
Cada base de clases tiene un especial sistema que se ha mantenido con atributos llamados surrogate. Todas las eventuales clases derivadas de una base clase heredan los atributos de surrogate. El surrogate evalúa para cada entidad en una clase que debe ser única, y no puede ser cambiada una vez definida .
En SIM, los surrogates juegan un papel central en la implementación ó generalización de jerarquías y de relaciones de entidades.

ATRIBUTOS

En SIM, una distinción es hecha entre atributos de valores de datos (DVA) y atributos de valores de entidad (EVA). Un atributo de valor de datos describe una propiedad de cada entidad en una clase por asociación de las entidades con un valor ó un multiconjunto de valores de un dominio de valores. La definición de DVA en SIM y de atributo en el modelo E-R son similares. Los valores de DVA, pueden ser puestos en una pantalla ó impresora, nombre y día de nacimiento de la clase persona son ejemplos de DVAs.
Un atributo de valor de entidad (EVA) describe una propiedad de cada entidad de una clase por relación de una entidad o entidades con otra clase o quizá con otra entidad en la misma clase. Un EVA representa una relación binaria entre la clase que le pertenece, el dominio y la clase de los puntos rango. En SIM usa las EVAs de relaciones de modelo entre entidades. Aquí, el valor de un EVA no puede ser puesto en un aparato de salida.
Un purista puede editar la distinción entre DVA y EVA, suponiendo clases base de enteros, strings, etc. se puede superar el tipo de declaraciones, explícitas del modelo requiriendo que estas sean declamadas ó pre-enumeradas subclases. Por ejemplo, el número -ID puede ser representado por una subclase de la clase base INTERGER con unas condiciones de rango apropiadas. Mientras una definición puramente funcional de todos los atributos puede ser utilizada antiestéticamente; se ha observado que muchos de los usuarios tienen diferente manera de entenderlo. Los tipos de datos explícitos en SIM naturalmente importados en su lenguaje de programa; se puede sentir que la aproximación intuitiva y más fácilmente entendida y es escogida por consideraciones históricas. SIM provee integridad completa referencial por mantenimiento automático de la inversa de cada declaración EVA y garantizando que un EVA este a la inversa que puedan estar sincronizadas en todos los tiempos. Por ejemplo, cuando una entidad es borrada, todas las referancias a otras entidades vía EVAs también son automáticamente borradas excepto que se tenga un restricción integra que pueda ser violada. Una inversa también puede ser explícitamente llamada por el usuario. Por ejemplo Advisees es el inverso de Advisor. En DML, el término Inverse (advisor) puede ser usado en cualquier contexto donde Advisees es permitido.
Una subclase hereda todos los atributos de todas éstas superclases en ésta jerarquía generalizada. En el ejemplo esquema, los atributos de Persona pueden ser vistos cómo una parte integral del estudiante, dado que cada estudiante debe ser una persona. Un atributo debe ser inmediato de la base clases de la subclase que se ha declarado. En DML, un atributo heredado de una subclase puede ser usado en cualquier contexto donde un atributo inmediato es permitido y viceversa. Por ejemplo, el estudiante ID es un atributo inmediato del estudiante mientras que nombre es un atributo del estudiante heredado de la persona. SIM permite referencias del Nombre Estudiante, Nombre de Persona, y Estudiante - y d de la persona; no tiene significado porque sólo los estudiantes tienen Estudiante -y d s. Los atributos son heredados de todos los niveles subordinados en una jerarquía generalizada, por ejemplo: Los asistentes de maestros heredan el Nombre de la persona vía Estudiante e Instructor.

En SIM, cada clase tiene una subclase y debe tener un especial atributo en un subpapel declarado en él. Por ejemplo, Profesión de Persona. Un subpapel es un caso especial de tipos enumerados y su valor establecido contiene los nombres de todas las subclases inmediatas de la clase en la cuál es usada. Los atributos del subpapel son mantenidos en el sistema y pueden ser únicamente leídos. Ellos incluyen en la lista destino de una obtención de consulta, proporcionando un método conveniente para recuperar simbólicamente todos los roles de los participantes en la entidad, y proporcionando una manera de control usando opciones de atributo de cómo las entidades pueden participar en jerarquías generalizadas.
Los tipos de atributos y las opciones son usadas en combinación para limitar los atributos de los valores y pueden ser entonces capturadas muchas estructuras relacionadas a la integridad y a los sistemas de bases de datos SIM.

TIPOS DE ATRIBUTO

Para DVAs, SIM proporciona un tipo de mecanismo similar al usarlo en Pascal. Los tipos pueden ser declarados en línea ó declarados y usados a través de nuestro esquema. Por ejemplo el Nombre de la Persona es una cadena de carácter de longitud de 30, mientras que el Estudant-Id ó el Estudiante y Empleado-id del Instructor están basados en él globalmente tipo declarado ID-Número. Los tipos deben ser basados en un sistema primitivo de tipos y pueden ser usados para restringir el rango de valores de DVA que pueden asumir por aplicación rangos apropiados de condiciones como fue hecho para el Número-ID en el ejemplo del esquema. Los sistemas proporcionados primitivos incluyen INTEGER, CHARACTER, NUMBER, (con decimales), REAL, BOOLEANO, STRING, DATE, TIME Y SIMBOLIC. De éstos mecanismos provee más latitud cuando se comparan tipos basados en cadenas como es permitido típicamente en PASCAL.

OPCIONES DE ATRIBUCION

REQUIIRED, UNIQUE, MV, DISTINCT y MAX son las opciones de atributo soportadas en SIM. REQUIRED implica que el valor de un atributo no puede ser nulo (nulo es usado para representar ya sea "inaplicable" ó "desconocido". UNIQUE implica que dos entidades de la clase no pueden tener un valor en común. Los valores Null son omitidos de las condiciones de unique.
MV indica que un atributo puede ser multi-valuado. Por default, los atributos son valuados simplemente. La opción DISTINCT es una agrupación múltiple que permite duplicados. MAX limita el número de valores de un MV y por default no tiene fronteras.
Atribuciones de clases. Las atribuciones de clases aplican sobre todo los miembros de la clase tomados como un todo. Por ejemplo, el Salario Máximo es una atribución de clase atribuida a un Instructor. Tales atributos pueden tener sólo un valor y son normalmente usados para capturar estadísticamente una formación

común a todos los miembros de la clase. Un diferente conjunto de atributos puede ser asociado con cada una de las clases con jerarquía generalizada, y deben obedecer reglas normales de herencia de atributos. Los Valores de los atributos de una clase pueden ser utilizados si ellos son constantes con la obtención de consultas DML. La implementación inicial de SIM requiere que los atributos de la clase sean valuados simplemente DVAs.

VISTA Y SEGURIDAD

La definición de vista en un modelo semántico posee problemas de desafío. En SIM, cada clase es un tipo nominal y las clases están fuertemente interrelacionadas. La salida de las consultas DML sigue un registro simple de estructura y no directamente refleja las interconexiones de entidades. Aquí una vista no puede ser definida como la salida de una obtención de consulta. Actualmente se diseñan extensiones a DDL y DML para vistas de soporte en bases de datos SIM. Sin embargo, los mecanismos de seguridad de SIM y las nociones DML de perspectiva y atributos de extensión grandemente eliminan la necesidad de las vistas .
La seguridad en SIM tiene dos componentes: Acceso y Permiso. Un acceso es especificando en la siguiente manera:

Access (access name) ON (class name) (attribute list)
(DML action) WHERE SELECTION expression

(attribute list) debe ser un subconjunto de clases con atributos inmediatos y la selección de la expresión debe involucrar sólo constantes ó atributos inmediatos de clase, (DML action) indica que cualquier clase puede ser recuperada, insertada, borrada, ó modificada.
Una asignación de Permiso es un acceso particular a un usuario dado. Una petición DML debe ser rechazada si se usa como un atributo de una clase que no este en la forma de (attribute list) al acceso seleccionado por el usuario.
Para una entidad en una clase C que sea visible al usuario, su código de usuario, así como las expresiones de selección asociadas con el acceso en el cuál todas las clases de antecesor de C tienen jerarquías. Este chequeo deberá ser llevado a cabo por el sistema independiente de cómo la clase es accesada ya sea directamente, a través de papales ó conversión a través de EVAs. Una especificación de acceso en una definición de Vista es **disguise**. Esto es en una manera tal que las clases que ya están definidas, no sean distribuídas y que nuevos tipos no sean distribuidos.

CLASES DE PERCEPCION

La noción de clases de percepción es de fundamental importancia en SIM DML. Está basada en la suposición de formulación de una consulta ya sea de

recuperación o de actualización, un usuario es principalmente interesado de una clase, llamada la perspectiva. Otras clases en la base de datos son vistas basadas en sus relaciones o en sus clases perspectivas. Usadas en combinación con otros hechos de SIM, la noción de clase perspectiva simplifica la consulta y permite a los usuarios diferentes intereses para aproximarse a bases de datos desde puntos de vista apropiados a sus necesidades.
Para un entendimiento de los atributos extendidos, consideremos una cuestión en la cuál, es una clase perspectiva.
Estudiante-id de Estudiante se refiere a un atributo inmediato. Nombre de Estudiante se refiere al atributo de herencia de la persona. Nombre del Instructor del Estudiante se refiere al Nombre de su instructor, y Profesor del curso - Enrolados- Estudiante se refiere a los instructores que enseñan los cursos en los cuáles están enrolados. Las últimas dos calificaciones son atribuidas al estudiante para propósitos de esta consulta.
Escogiendo una clase perspectiva apropiada, los usuarios pueden entonces establecer sus propias consultas DML en términos relevantes a sus necesidades Por ejemplo, varios usuarios pueden interpretar la palabra "Physics" diferentemente. Un usuario involucrado con departamentos de universidad pueden ser "Physics" como un valor para un Nombre de un Departamento, alguno en el departamento de personal puede interpretar como el Nombre asociado al Departamento de un instructor, y un trabajador en una oficina de Admisiones puede pensar como el nombre de un Departamento de Estudiantes. De hecho, todos los tres usuarios están viendo la misma pieza de almacenamiento de información pero en manera "semánticamente relevantes" para sus usos. La noción de las clases perspectivas pueden ser combinadas con jerarquías de generalización para simplificar la formación de consultas. Por ejemplo, consideremos la pregunta "Escribe el nombre de cada estudiante y el nombre de su profesor, si hay ?.
En DHL, esta pregunta es expresada como:

FROM Estudiante RETRIEVE, Nombre, Nombre de Profesor,

Los nombres de personas que no son estudiantes no pueden ser escritas. Sin embargo, si un estudiante no tiene un profesor, SIM seleccionará y escribirá su nombre a un valor nulo para el nombre de su profesor. La perspección de esta consulta es si el Estudiante y el Nombre del Profesor se refiere a un atributo extendido. La misma información puede ser recuperada escogiendo Persona en la clase perspectiva, pero la consulta será más complicada:

FORM PERSONA RETRIEVE Nombre ,Nombre de Profesor AS Estudiante
WHERE Profesión = Estudiante.

LAS CLAUSULAS

AS y WHERE pueden ser añadidos para excluir Personas que no son Estudiantes. Algunas veces puede ser necesario formar consultas con más de

una clase perspectiva; por ejemplo: Todos los pares de estudiantes que han tomado el mismo número de curso.
SIM proporciona esto como una facilidad y la formulación de consultas es llena de consultas de múltiple perspectiva. Las clases de múltiples perspectivas están relacionadas una con otra por valores unidos, los cuales establecen relaciones dinámicas entre clases. Se recomienda fuertemente el uso de UVAs para valores unidos dado que ellos representan un esquema definido, estático, eficiente y natural para establecer relaciones.

SINTAXIS DE OBTENCION DE CONSULTAS

Una obtención de consulta DML es expresada en la forma

[FROM <porspective class list>]
RETRIEVE [TABLE [DISTINCT] ! STRUCTURE] <TARGER LIST>
[ORDER By <Order list >]
[WHERE < Selection expression>]

(perspective class list) es la lista perspectiva de clases para una consulta con variables opcionales de referencia asociadas. (target list) y (order list) son una lista de expresiones hechas de constantes, Inmediatas, herencia y atributos extendidos o de perspectivas, y agregados con otras funciones aplicadas con tales atributos.

UNION Y VARIABLES DE RANGO

La semántica de las consultas de SIM son entendidos en términos de loops iterativos anidados similares a SATA PLEX. Todas las ocurrencias de una clase perspectiva en una consulta son "unidas" a una variable de rango. (loop). Similarmente todas las ocurrencias de una Idénticamente calificada EVA o multivaluada DVA . También son unidas a variables de un rango. Consideremos la siguiente consulta:

RETRIEVE Nombre de Estudiante
Titulo de Cursos Tomados de Estudiante
Nombre de Profesores de cursos- Tomados de Estudiante
WHERE SOC-Sec-No- de Estudiante = 80207370

Para el Estudiante con Soc-Sec-No 80207370, ésta consulta imprimirá su nombre y para cada curso que él esté tomando, su titulo, los créditos, y el nombre de los instructores quienes le enseñan esto. Esto es posible por solo cinco ocurrencias de la literal "Estudiante" y todas las tres ocurrencias de la literal "Cursos Tomados " que son unidos a sus respectivas variables de rango.
Uniones implícitas de nombres se rompen con pocas construcciones especiales como son funciones de agregación, clausura nominada de cuantificadores. Cuando se necesitan, variables nominadas de rango pueden también ser

establecidas explícitamente en una clase, EVA ó multivaluadas DVA con el constructor CALLED, por ejemplo:

```
RETRIEVE Nombre de Instructor
 WHERE Gpa de  Adviss ( z.o and
        Gpa de Advises  CALLED Buen Estudiant ) 3.5
```

regresa el nombre de todos los instructores que enseñan a los estudiantes que lo están haciendo bien y los estudiantes que lo están haciendo pobremente. El nombre de la variable rango es requerido porque la unión normal de la regla asegura que la misma expresión de Selección sin el constructor CALL regresará a todos las Entidades.

SEMANTICA DE OBTENCION DE CONSULTAS

La calificación de reglas de unión de SIM, tomadas juntas elevan el concepto de la consulta de árbol. La consulta de árboles generada por una consulta particular es convertida en unir serie de loops anidado que SIM utiliza para ejecutor la consulta. Cada loop anidado es bajo el control de una variable de rango, y el más exterior de los loops es controlado por la variable de rango unida a la clase perspectiva si no es alterada por el optimizador. La secuencia de anidación de éstos loops determina el orden en la cual las consultas resultantes son regresadas al usuario. Estos loops anidados tienen la misma estructura lógica que puede programar una aplicación convencional escrita para completar el mismo resultado usando un sistema de archivo en un registro orientado a DBMS.

FUNCIONES DE AGREGACION

En SIM las funciones de agregación son específicamente y naturalmente delimitados por su escenario en una calificación. Considerando los siguientes ejemplos:

AVG (Salario de Instructor)
AVG (Salario de Instructores, empleados) de Departamento,
Count (Profesores de Cursos Tomados) de Estudiante.

El primero da el salario promedio de todos los instructores en la base de datos, el segundo da el salario promedio de los empleados por cada departamento, el cuál es dinámicamente derivado al atributo del departamento y el tercero da para cada estudiante el número de maestros de todos los cursos en los que están inscritos ó están tomando cursos.
Los cuantificadores (ALL, SOME y NO) siguen una sintaxis similar.

VARIOS

El DML también soporta cuantificadores, patrones de chequeo y arreglos de operadores y funicones primitivas. Los valores Nulos son tratados uniformemente

en la evaluación de expresiones, y SIM sigue la lógica de tres valores (True, False y Null), verdadero, Falso y Nulo. Posteriormente se muestra el poder y el uso de estos SIM DML.

1.- Inserte Luis Mendoza como un estudiante y posicionelo en Algebra 1.
Inserte Estudiante)Nombre: = "Luis Mendoza",
Soc- Sec- No: = 80207379
Courses-Tomados: INCLUDE Course WIHT (Título = "Algebra Y")).

2.- Hacer que José Luis Mendoza sea un Instructor.
Insert Instructor
FROM Persona WHERE Nombre = "" Luis Mendoza "
(Empleado - Nbr: = 1728).

3.- Deje que Luis Mendoza quede en Algebra y deje a Armando Huerta como su instructor .
MODIFY Estudiante
Curso - Tomados: = EXCLUDE Cursos - Tomados
WITH (Título = " Algebra I "
Advisor : = Instructor WITH (Nombre = "Armando Huerta"))
WHERE Nombre de Estudiante = " Luis Mendoza"

4.- Si un instructor enseña más de 3 cursos e instruye a estudiantes de otros departamentos, darle un 10% de aumento.
MODIFY Instructor (Salario: = 1.1x Salario)
Asignar- Departamento NEO
SOME (Mejor- Departamento de Instructores).

5.- Encuentra el mínimo de cursos que deben de ser terminados antes de tomar dinámica cuántica .

FROM Course
RETRIEVE COUNT DISTINCT (TRANSITIVE (Prerequisito))
WHERE Titulo = dinámica cuántica

6. Imprima el nombre de cada instructor que instruye algunos estudiantes del departamento de Física y los cursos que el enseña, si los hay:
RETRIEVE Nombre del instructor, Título de cursos - enseñados
WHERE Nombre del mejor - departamento de instructores = Física
7. Imprima el estudiante, los pares de instructores donde el estudiante es más viejo que el instructor y que el instructor no tiene un asistente de enseñanza y que no es el Instructor del estudiante.
FROM Estudiante, instructor
RETRIEVE Nombre de estudiante, nombre de instructor

WHERE Dianac de estudiante de instructor AND ADVISOR de estudiante NEO Instructor AND NOT Instructor ISA Asistente - Enseñanza.

CONSIDERACIONES DE IMPLEMENTACION

Objetivos
En adición a sus funciones objetivos, SIM trabaja específicamente con objetivos de implementación .

Ambiente
SIM ha sido inicialmente implementado sobre el sistema Unisys lines de estructuras series A . Todas las series A son compatibles con códigos de objeto y varían en sistemas de nivel de entrada hacia estructuras de gran escala . Los principales datos y módulos de Software de la base de datos de SIM deben residir sobre una serie A y los datos deben ser accesibles a través de estaciones de trabajo así como terminales conectadas hacia la computadora madre . La heterogeneidad y la distribución de los accesos de datos también debe ser direccionada.

Compatibilidad.
Un mayor objetivo de SIM es ser totalmente compatible e integrada con los sistemas administradores de datos (DMSII), una estructura DBMS en la cual es utilizada casi toda la serie A . Las actuales bases de datos DMSII deben ver el SIM como funcional y adicional en sus aplicaciones de datos existentes y códigos para reservarlos. Así una trayectoria de migración debe ser proporcionada para una base de datos DBMSII que pueda ser convertida en una base de datos SIM .

Desarrollo
Los sistemas de serie A primeramente soportados con aplicación comercial que cumple un amplio rango de necesidades de procesamiento de datos . Al final de este rango los sistemas que requieren altas transacciones de velocidad (cien transacciones por segundo o mas) y los sistemas que accesan grandes cantidades de datos (10 gigabytes de datos o mas) SIM también proporciona desarrollos a nivel de producción para el rango completo de aplicación de sistemas encontrados en la serie A, incluyendo aquellos superiores.

Independencia de Datos
SIM también debe proveer el mas alto grado de independencia de datos posible de manera que los cambios físicos de las bases de datos pueden ser hechos sin impactar en las aplicaciones existentes. La lógica de los cambios de las bases de datos no deben afectar los programas de consulta aunque sean relevantes y cuando se necesiten, las consultas, la reorganización y reoptimización deben ser automáticas.

Accesos Heterogéneos de datos

Los objetivos iniciales de SIM son para proporcionar accesos simples de datos en los cuales residen las bases de datos de DMSII y simples archivos de datos. Sin embargo, su arquitectura debe ser acomodada a otras fuentes y tipos de datos incluyendo estructuras de datos complejas, datos transitorios tales como interfaces de procesos y residencia de datos en hosts ajenos. La meta de este objetivo es asegurar que ambos el modelo y la implementación de la arquitectura de SIM sean lo suficientemente flexibles para permitir una organización para usar modelos uniformes con los cuales pueden representar y accesar toda esta información .

Arquitectura funcional

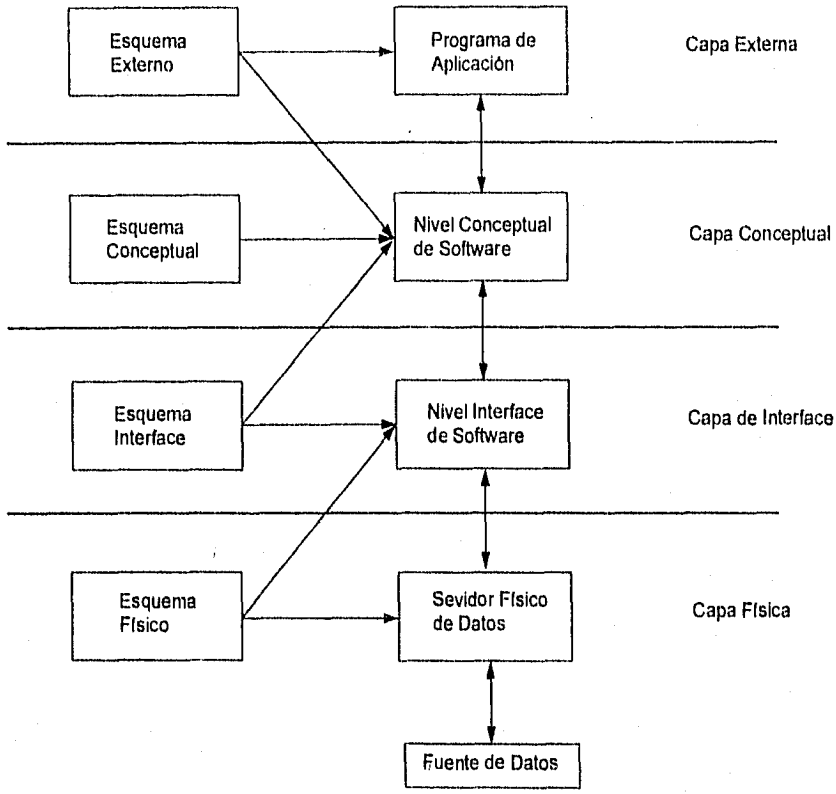
La Arquitectura Funcional básica utilizada para SIM es una versión mejorada del ANSI / SPARC . SIM emplea las capas externas y conceptuales, pero la capa interna propuesta ha sido puesta en una capa de interface y en una fase física.

La arquitectura funcional de SIM es una versión mejorada del ANSI/SPARC en la cual el nivel más bajo, la capa interna ha sido puesta en una interface y en una fase física. Cada capa posee un esquema asociado a los componentes DBMS del Software .

Cada capa funcional posee un esquema la cual define la base de datos de un nivel sucesivo mínimo . Cada capa posee también cierto software DBMS el cual es involucrado en crear mantener o accesar la base de datos. Cabe mencionar que la arquitectura ANSI/SPARC define una información de mapeo la cual existe entre cada capa .

La ventaja primaria de esta arquitectura es que permite a un servidor de datos existente proporcionar tantas funcionalidades como esto puede ser para una fuente particular de datos. En muchos casos la capa física puede ser proporcionada por un sistema de software ya existente. Las capas externas y conceptuales de SYM son idénticas sin importar el tipo de datos para la fuente utilizada. Aunque la interface SIM debe ser añadida para cada tipo de fuente de datos, si necesita la implementación de funcionalidad no proporcionada en la capa física. Esta aproximación minimiza la cantidad de ajustes que deben ser hechos para cada tipo de fuente de datos que nosotros queramos accesar a una base de datos SIM.

El resultado de la arquitectura se presenta a continuación.



Capa externa
El propósito de esta capa es proporcionar al usuario final una aplicación de acceso a la base de datos. Cada usuario o programa de aplicación que accese a SIM estará relativa a los esquemas externos. Con el SIM los esquemas externos predeclarados no son necesarios para propósitos de seguridad dado que los esquemas conceptuales ofrecen predicados declarativos orientados a mecanismos que causan seguridad propia para ser mejorada automáticamente .

Capa conceptual . A este nivel la base de datos total es definida vía un esquema conceptual . Un esquema conceptual para una base de datos SIM contiene ambas definiciones estructurales (por ejemplo generalización de jerarquías, sus atributos

y relaciones) y definiciones imperativas por ejemplo: seguridad y restricciones de integridad . Sin embargo, para proporcionar independencia de datos, el esquema conceptual es completamente libre de referencias para la distribución de registros u otros detalles físicos de almacenamiento .

Capa de interface

A este nivel, un esquema de interface es generado para cada base de datos del programa utilitario SIM . El esquema interface es definido en términos de un modelo más simple y de menor nivel que el modelo semántico utilizado para el esquema conceptual. El modelo consiste de componentes lógicos fundamentales. (LUC) en el cual cada uno de los cuatro tipos: registros, campos de registros, índices y relaciones estos tipos LUC son escogidos porque:

- Los componentes del esquema conceptual son fácilmente mapeados en el LUC de los objetos de estos tipos .
- Los objetos LUC poseen suficiente información física para permitir consultas efectivas y optimizadas .
- El procedimiento de consultas del software así como los niveles conceptuales solo necesitan generar operaciones sobre un conjunto limitado de componentes estructurales, por ejemplo: limitadas cuando se comparan con el número de componentes físicos que pueden ser usados .
- Colectivamente, estos tipos LUC pueden describir y por lo tanto, los objetos LUC pueden ser mapeados para cualquier tipo de almacenamiento orientado hacia el registro .

CONCLUSION Y DESARROLLOS FUTUROS

Se describe el SIM como un sistema de bases de datos basado en un modelo semántico de datos. SIM proporciona un punto de vista natural de datos moviéndolos a través de la simplicidad notacional por modelamiento con un conjunto completo mínimo de constructores .

Entidades, generalización, jerarquías , relaciones interobjeto del esquema definido y restricciones de integridad son las claves conceptuales del modelo . El DML de este sistema es diseñado para tomar ventaja y directamente soportar características. Las nociones del DML de perspectiva y calificación por EVAS son completamente naturales del esquema de definición de estas ventajas .

La experiencia con un gran número de análisis de bases de datos es testimonio de la potencia y utilidad de los conceptos mencionados anteriormente . El estado actual del diccionario de datos ADDS es por si mismo una base de datos SIM . Este consiste de trece clases base, 209 subclases, 39 EVA - Pares inversos, 530 DVAS y en su nivel más profundo, una jerarquía que representa 5 niveles de generalización .

Actualmente se esta trabajando en varias extensiones del modelo . El trabajo progresa incluyendo el diseño de mecanismos de vista, atributos derivados, ordenamiento de sistemas mantenidos de clase y datos temporales de EVAS,

eficientes algoritmos para varias categorías de restricciones de integridad y experimentos en cuantificación de falta de naturalidad y facilidad de uso del DDL y los conceptos DML .

4.5 GEMSTONE

Se han desarrollado los resultados de la base de datos orientada a objetos GemStone, la cual soporta un modelo de objetos similar a el de Smalltalk-80. Se han hecho resúmenes de las metas y requerimientos para el sistema: un modelo de datos que capture el comportamiento semántico, límites no artificiales en el número o tamaño de la base de datos de objetos; bases de datos agradables (transacciones, recuperación, autorizaciones) y un desarrollo interactivo en el medio ambiente. De los lenguajes orientados a objetos, SmallTalk en particular, responde a algunos de esos requerimientos.

INTRODUCCIÓN

El sistema de base de datos GemStone es el resultado del desarrollo de un proyecto que empezó hace tres años en Servio. Su objetivo fue la mezcla de los conceptos de lenguajes orientados a objetos con los sistemas de bases de datos. GemStone proporciona un lenguaje de bases de datos orientado a objetos llamado OPAL, el cual es usado para la definición de datos , manipulación de datos y computación en general.
La premisa es de una combinación de capacidades de un lenguaje orientado a objetos con el almacenamiento de manejador de funciones de un sistema manejador de datos tradicional que resultaría un sistema que ofrece mas allá de reducciones en el esfuerzo del desarrollo de ejecuciones. La extensa facilidad de tipificación de datos del sistema facilitaría el almacenamiento de información que no pertenece normalmente a las relaciones. Además se cree que un lenguaje orientado a objetos puede ser completamente suficiente para el diseño del manejo de la base de datos, acceso a la base de datos, y aplicaciones.

METAS Y REQUERIMIENTOS

El sistema puede tener un modelo de datos que soporta la definición de nuevos tipos de datos, mas bien que los programadores se obliguen a usar un conjunto fijo de tipos predefinidos.
El objetivo es que el comportamiento del modelo no es justamente la estructura de la entidad en el mundo real. Por lo tanto se debe ser capaz para empacar el comportamiento con la estructura y crear nuevos tipos de datos. Para obtener una ejecución razonable para tal sistema, la colección de constructores debe ser lo suficiente rica para la mayoría de los datos que tienen equitativamente implementaciones directas. En particular, se debe ser capaz de capturar muchas relaciones, colecciones y secuencias directamente. Para un sistema fácil de

usar, se debe ser capaz de anidar las operaciones de estructura en niveles arbitrarios y usar tipos de datos previamente definidos como bloques de construcción para otros tipos . GemStone debe tener funciones de administración para monitorear el comportamiento del sistema de los respaldos y de la recuperación por fallas, quitando y poniendo usuarios y alterando los privilegios de los usuarios.

BASES DE DATOS AGRADABLES

GemStone es primero un sistema de datos, asl que debe permitir acceso compartido a datos persistentes en un ambiente multiusuario. Debe tener un estable soporte de almacenamiento de datos en el disco mientras provee la transparencia de lugar para la aplicación del programador en el movimiento de objetos entre la memoria principal y la secundaria.
GemStone debe proveer una propiedad de datos, y solicitud de propiedad para autorizar , compartir los datos con otros usuarios. Cada sesión de datos deberá aparecer para tener un control completo de una sesión consistente de la base, aun cuando los usuarios estén trabajando conscientemente, y debe proveer un mecanismo de transacción para usar un grupo de cambios atómicamente. Los usuarios deberán ser capaces de solicitar la duplicación de datos críticos en caso de falla del sistema.
GemStone debe soportar estructuras de almacenamiento auxillar que proveen accesos aitemativos a los datos y debe dar control a los usuarios sobre la agrupación física de objetos para incrementar la eficiencia de rutas especificas de acceso. La unión de datos y tamaños deberá determinarse solo por el almacenamiento secundario, uno por las restricciones de la memoria principal o de la definición de los datos. Así, los datos en un registro serán de longitud variable pero no fijados en su unión superior. Grupos de objetos como arreglos no deben tener enlaces con el número de elementos, similarmente, el número total de objetos en un sistema de datos no debe manejar objetos grandes y pequeños con razonable eficiencia.

MEDIO AMBIENTE DE LA PROGRAMACIÓN

GemStone tiene almenos las siguientes herramientas y propiedades para desarrollo:
1: Fases interactivas por definición de nuevas bases de datos de objetos, escribiendo rutinas de OPAL y ejecutando (QUERIES) adecuadas en OPAL.
2: Empaquetando por ventanas sobre los cuales las interfaces de usuario pueden ser construidas.
3: Procedimientos de interface a lenguajes convencionales como C y Pascal.

VENTAJAS DE UN MODELO ORIENTADO A OBJETOS.

Durante las etapas de investigación del proyecto GemStone, se desarrollaron la mayoría de las preguntas del lenguaje que fueran difíciles en capacidades del procedimiento. Dados los problemas para proveer extensiones de procedimiento y educación del mercado y para complementar un nuevo lenguaje, decidimos usar un lenguaje Orientado a Objetos ya existente. SmallTalk-80 como base del desarrollo del producto. Se realizaron extensiones de SmallTalk en las áreas de accesos asociados para preguntas, estructuras-básicas de almacén, escritura y soporte para ambiente multiusuarios.

PODER DEL MODELO

GemStone ofrece modelamiento de objetos complejos y relaciona directamente y organiza tipos de datos en una herencia jerarquizada. Una entidad simple es modelada como objeto simple no como múltiples TUPLAS dispersas o como varias relaciones. Las propiedades de las entidades no necesitan ser valores de datos, pero pueden ser otras entidades de complejidad arbitraria. El componente utilizado de un objeto empleado no necesita ser solo un texto cadena. En GemStone puede estructurar el objeto, con sus propios componentes por número de calle y ciudad así como su conducta definida.
GemStone directamente soporta valores establecidos sin el código requerido en el modelo relacional. Por lo tanto los grupos pueden tener objetos arbitrarios como elementos y no necesitan ser homogéneos. GemStone da la independencia física a los datos de bases relaciónales sin limitaciones sobre una potencia de modelado.

IDENTIDAD DEL OBJETO

GemStone provee identidad de objetos. Los datos conservan su identidad a través de cambios arbitrarios en su propio estado.
Entidades con información común pueden ser modeladas como dos objetos con información común compartida. Esta comparición reduce anomalías de autorización que existen en modelos relaciónales de datos. En el modelo relacional, las propiedades de una entidad deben ser suficientes para distinguirlas de otras. Para referir una entidad de otra, debe haber campos que identifican unívocamente cada uno de ellos. (Algunas extensiones a los modelos relaciónales incorporan formas de identidad). Inmutabilidad y exclusividad son ideales raramente presentes en el mundo real. Podemos referirnos a departamentos por nombre, pero si el nombre del departamento cambia? (sabemos que todos los objetos son independientes y que cada objeto es una propiedad o depende de algo, que puede ser de otra manera administrado el almacenamiento.

COMPORTAMIENTO DEL MODELO

GemStone ofrece simulación de comportamiento en entidades del mundo real. Los comandos de manipulación de datos en sistemas convencionales están orientados a través de representaciones de máquina: "Campos modificados", Insertar tuplas". Para un sistema de administración de oficina, algunas aplicaciones pueden reservar un espacio: En un sistema como: de base de datos, cada aplicación puede contener DML (Lenguaje de Manejo de Datos) comandos para probar la disponibilidad del espacio, insertar cambios a registros e indicar reservación y quizá crear otro registro. Cambios en la estructura de la Base de Datos puede requerir ubicar y modificar cada aplicación que haga uso de la Base de Datos . En GemStone, un mensaje Reserve Room puede ser definido a una fecha y un tiempo como parámetro y hacer todos los chequeos necesarios y actualizar la base de datos a reservar en el espacio.
El método OPAL que implanta el mensaje puede ejecutar cualquier número de preguntas de bases de datos y actualizar con muchas ventajas. Las aplicaciones son mas concisas: Enviando un mensaje de lugar a muchos cambios en las operaciones de Bases de Datos. El código es mas útil, como aplicación que reserva un espacio usa exactamente el mismo procedimiento - el método asociado con el mensaje espacio reservado. El escenario de cambios requerido por alteraciones a la estructura de un tipo es limitado por los métodos para el tipo. Así los mensajes pueden proteger la integridad de la base de datos por consistencia en sus métodos.

CLASES

La estructura de clases de GemStone acelera el desarrollo de aplicaciones en varias maneras. GemStone viene con un gran complemento de clases utilizadas en tipos de datos. Las definiciones de clases son los análogos a esquemas en otros sistemas de bases de datos, pero las clases además empacan aplicaciones con la estructura para el comportamiento de la encapsulación. Así la definición del mensaje facilita el mecanismo con clases que requieren modelos extensibles de datos. GemStone incluye jerarquía de clases. Mientras una da ayuda a organización de datos, la jerarquía ayuda a organización de clases. El mecanismo de subclases permite a un esquema de bases de datos capturar similitudes entre varias clases de entidades que no son totalmente idénticas en estructura o comportamiento. Las subclases además proveen medios de manejo especial de clases sin confundir la definición de caja normal.

ASOCIACIÓN DE TIPOS CON OBJETOS

Diferente a la mayoría de lenguajes de programación que usan tipos de datos abstractos, Smalltalk asocia tipos con valores, no los números manejan los valores. Objetos escritos mas que normales tienen propiedades para procesamiento de consultas.

Esperamos establecer diseños de bases de datos para aplicación en modelos con dominio que previamente puedan ser **SHIED AWAY** debido a la complejidad o falta de estructura regular. Sin embargo, el modelamiento es una empresa para la primera vez y es muy difícil tomar una base previa construida para una área ya modelada pero aun no computarizada.
El modelo básico para registro financiero fue hecho miles de años antes. La estructura de información involucrada es de tal manera que es fácilmente establecida en bases de registro comunes. El desarrollo de un esquema basado en la definición de esquema, aplicación de la escritura, datos de población, y chequeo es razonable. No así el objetivo de CAD que fue modelado por primera vez o una base de datos que soporta un sistema experto. El área de aplicación no había sido modelada antes, y habrá muchas interacciones del esquema de bases de datos antes de la aplicación madura. Estando lista para escribir rutinas de Bases de datos sin tener completamente la especificación de la estructura y el comportamiento de cada entidad de clases puede ser de gran ventaja. Después, cuando el modelo este establecido, puede ser escrito para ser asociado con campos para la integridad o eficiencia.
No por asociación de tipos con variables, casos no anticipados (una compañía de autos podría ser asignada a un departamento como un empleado) puede ser mas fácilmente manejado. Una rutina (método) hace suposiciones acerca del protocolo de sus argumentos, no de sus estructuras internas.
Tales rutinas son robustas en la cara de nuevas clases, si cada objeto responde al mensaje PRINTSTRING para retomar como cadena misma, entonces se puede escribir un método OPAL para SET que imprima una representación en cadena de todos sus elementos no importando sus clases.

LENGUAJE UNIFICADO

OPAL es mucho mas potente que cualquier lenguaje de manipulación de datos. Es computacionalmente completo, con asignamientos y construcciones para flujo de control. Casi toda la computación requerida en una aplicación puede ser escrita dentro de OPAL. Esta habilidad ayuda a prevenir el problema de perdida de impedancia (impedance mismatch, donde la información debe pasar dentro de dos lenguajes que son semántica y estructuralmente diferentes, tales como un sublenguaje declarativo de datos y un lenguaje imperativo de propósito general. GemStone uniformiza el acceso a todos los sistemas objetos y funciones, utilizando los mismos mecanismos que un sistema regular de datos objeto.

REGRESANDO SMALLTALK DENTRO DE UN DBMS

Smalltalk es un sistema de usuario simple, basado en la memoria, de procesador simple. Pero no cumple con los requerimientos de un sistema de bases de datos. Mientas que Smalltalk provee un potente uso de interfaces y un desarrollo de herramientas para aplicación, esta orientado a usuarios simples de estación de

trabajo. Para satisfacer los requerimientos de un sistema de bases de datos se han añadido las siguientes mejoras.

SOPORTE DE UN MULTI-USUARIO EN UN AMBIENTE BASADO EN DISCOS

El hecho de que este basado en discos no significa simplemente la paginación de la memoria principal de un disco y de un sobreflujo. La base de datos deberá ser inteligente acerca de los objetos establecidos entre el disco y la memoria. Y deberá tratar con grupos de objetos accesados junto o en las mismas paginas del disco, tratando de anticipar cuales objetos en la memoria son similares para usarse otra vez, rápidamente y organizar estas consultas de procesamiento para minimizar trafico en el disco.
Dado que los datos de GemStone están compartidos por múltiples usuarios, el sistema debe de proveer accesos variados. Cada usuario deberá ver una versión consistente de la base de datos aun cuando otro usuario lo este usando simultáneamente. Dado que un usuario puede hacer cambios que no están establecidos permanentemente ala base de datos, GemStone deberá soportar alguna noción de lugares de trabajo múltiples en los cuales, los cambios propuestos en la base de datos puedan ser posteriormente aceptados o descartados. Un requerimiento relacionado es la administración de múltiples espacios nominales. Smalltalk supone un usuario simple por imagen y así provee un nombre del espacio global simple. Parcialmente puede estar relacionado o no relacionado con aplicaciones que pueden estar bajo desarrollo de una base de datos simple a un tiempo. No es razonable esperar que ya sea un usuario que comparte un nombre de espacio global simple, o bien, en el otro extremo, que el nombre del usuario este desunido.
Actualmente las implementaciones de Smalltalk usan un procesador simple para el display por procesamiento o para el administrador de objetos. Se espera en GemStone el soporte múltiple interactivo de las aplicaciones. Aquí no parece ser muy sabio el uso de utilizar procesadores para la administración secundaria de la información así como para el proceso en display como interface final de usuario. Se puede observar que la administración del almacenamiento y que las funciones del usuario en interface en GemStone deberán ser separadas para utilizarse como procesos separados o procesadores separados.

INTEGRIDAD DE DATOS

Varias clases de fallas (programas, procesadores o medias) y violaciones (consistencias, accesos, escrituras) pueden comprometer la validad y la integridad de una base de datos. Un sistema de base de datos debe entonces ser capaz de tomar en cuenta con cualquier falla restableciendo la base de datos en un estado consistente, mientras minimiza la cantidad de perdida computacional. Se debe prever también violaciones de que esto ocurra.
Por fallas del programa se debe entender un programa de aplicación que puede fallar por completo, es decir, causando un error en el tiempo de corrida. Si el

programa falla después de algunas actualizaciones que se hayan hecho a la base de datos, esta puede entonces quedar en un estado no esperado. Los sistemas de bases de datos proveen para actualizaciones múltiples que sean efectuadas atómicamente (en todos o de ninguna manera) a través del uso de transacciones. Una transacción es usada para marcar una sección del proceso de manera que los cambios hechos sean permanentes, o bien, ninguno sea permanente; ya sea que la transacción se de por buena o aborte.
Por fallas del procesador se debe entender que el procesador que maneja GemStone de administración del almacenamiento falle. Para tales fallas la base de datos deberá mantenerse intacta. La recuperación del programa y del procesador de la falla implica que tengan copias maestras de los objetos en una memoria secundaria que debe ser actualizada cuidadosamente. Adicionalmente un buen sistema de base de datos deberá ser robusto lo suficientemente para tolerar fallas adicionales durante el periodo de recuperación.
Por fallas de media se debe entender el daño de los equipos de almacenamiento secundario que pueden causar que los datos se confirmen o se pierdan. No se tienen estrategias que puedan proveer la completa protección contra las fallas en media. Se trato en GemStone proveer periódicos y dinámicos respaldos o duplicación de la información sensitiva. Por duplicación dinámica se entiende mantener varias o múltiples copias alineadas a la base de datos, las cuales son actualizadas durante cada transacción.
Regresando a las violaciones, la consistencia de la base de datos puede ser violada siempre y cuando las transacciones de los usuarios múltiples dejan sus actualizaciones. Gemstone debe soportar la serialización de transacciones esto quiere decir que el efecto neto de transacciones concurrente de la base de datos puede ser equivalente a la ejecución en serie de varias de estas transacciones. La integridad de una base de datos también puede ser violada si el usuario tiene acceso a los datos que el o ella debe no ser permitida para verla. En Smalltalk, todos los objetos son disponibles para el usuario. GemStone debe asignar una propiedad única para cada objeto, y darle al propietario de un objeto la habilidad de accesarlo a otros.
La integridad de restricción, tal como una llave de integridad referencial son aserciones que a priori excluyen ciertos estados de la base de datos. Siempre sobre un juicio; ya sea que la base de datos deba checar estas restricciones después de cada transición, o bien, que la aplicación de programador debe ser responsable para preservar la consistencia en cada transacción. El curso es mas útil, pero muchas veces muy caro. Como mínimo, una base de datos debe soportar extensiones que requieran subpartes de una entidad o colección que pertenezca a una cierta clase. Se observa que la integridad referencial viene gratis en GemStone. La referencia a un objeto directamente a otro objeto, no con un nombre para un objeto. La referencia no puede ser creada si el otro objeto no existe. Es decir no hay referencias dangling.

ESPACIO LARGO DE OBJETO

GemStone debe guardar grandes números de objetos y objetos de gran tamaño en sus memorias. Las primeras implementaciones de Smalltalk-80 tienen un limite de 2^{15} objetos, 2^{15} variables en instancia en cualquier objeto 2^{20} total de palabras en memoria. Mas recientemente las implementaciones han incrementado estos limites, pero todavía se usan las mismas técnicas para representar y manejar objetos (OOPs) de gran tamaño. Los objetos grandes basados en discos requieren nuevas técnicas de almacenamiento. Algunos objetos serán tan largos para entrar en cualquier memoria, que deben ser compaginados.
Mientras que las implementaciones en la memoria virtual de objetos largos, se puede ver que debemos estar lejos de representaciones lineales de objetos grandes. Los objetos que requieren que sean guardados en paginas de disco o bien establecidos contiguamente en una memoria secundaria o aun en una memoria virtual conducirán a una fragmentación no aceptable o muy cara de la compactación. En Smalltalk, para "crear" un objeto tal como un arreglo nuevo, o un objeto grande cuando es creado y los contenidos de los objetos pequeños son copiados en el. Se pretende que el tiempo requerido para actualizar o extender el objeto sea proporcional al tamaño de la actualización o la extensión, no al tamaño del objeto que ya se ha actualizado. También se puede observar que en Smalltalk el repertorio de la representación básica de almacenamiento fue inadecuada para soportar colecciones grandes no ordenadas. Teniendo un mapa como una colección en una representación ordenada que impone restricciones artificiales. Por lo tanto, GemStone necesita un tipo básico de almacenamiento para colecciones no ordenadas.
Finalmente buscando una gran colección por un barrido secuencial que nos de un comportamiento no aceptado en un objeto basado en disco. La búsqueda para elementos deberá ser al menos logarítmica en el tamaño de la colección más que lineal. Por lo tanto, GemStone deberá soportar accesos asociados a elementos de grandes colecciones: deberá alimentar representaciones almacenadas y estructuras auxiliares para soportar la ubicación de elementos por su tamaño, por su estado interno. Estos requerimientos refuerzan la necesidad de escribir colecciones o variables de instancia. Para indexar una colección E de empleados en un valor de salario, por ejemplo una variable, el sistema necesita asegurar que cada elemento en E tiene una entrada en salario. Por lo tanto, si tal índice es consultado en rango de soporte en salario, el sistema necesita una declaración que todo el salario deberán ser comparables al total del orden establecido. Además con el soporte del almacenamiento por nivel asociado al acceso, OPAL debe tener unas construcciones de lenguaje que permitan el acceso asociado.

MANEJO DEL ALMACENAMIENTO FÍSICO

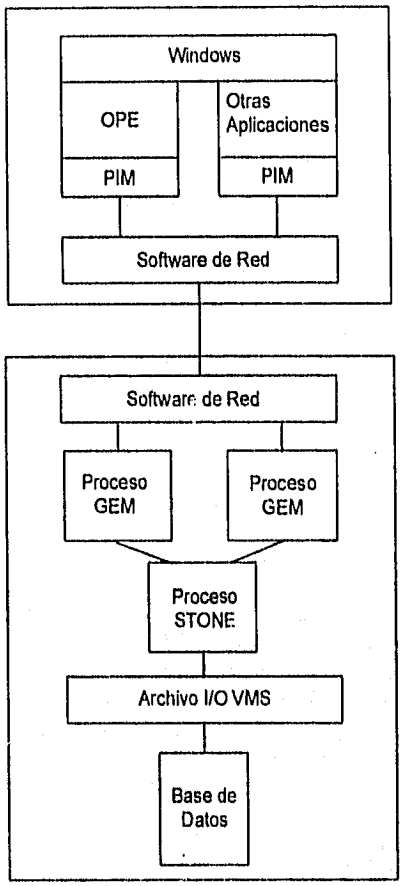
GemStone debe proveer propiedades para el manejo del establecimiento físico de objetos en el disco. Smalltalk es un sistema de memoria residente, así que no hay mucha necesidad para decir donde un objeto va. El administrador de la base de datos, o el programador de la aplicación savy, deberá ser capaz de avisar a GemStone que ciertos objetos serán algunas veces usados juntos, y que deben ser enclaustrados en el disco. El administrador deberá ser capaz de tomar objetos fuera de línea , digamos archivados y traerlos a la línea final. Finalmente, hay objetos que nunca están explícitamente borrados, el sistema deberá ser responsable para recuperarlos del espacio usado por objetos no referidos. (Una alternativa de suponer que un objeto permanente nunca es borrado, y que los objetos no referidos en el estado actual de la base de datos deberán ser cambiados a un almacenamiento de archivo).

ACCESOS DE OTROS SISTEMAS

Mientras que OPAL va mucho más allá que un lenguaje convencional de bases de datos, su objetivo principal es el de proveer un lenguaje para la aplicación de bases de datos de programación. Sin embargo, aquí en GemStone se pretende concentrar los esfuerzos iniciales en aspectos de manejo de almacén, más que en uso de interfaces. Por lo tanto, GemStone provee accesos para la facilidad de otros lenguajes de programación. Otra de las metas es la de soportar una aplicación del ambiente de desarrollo de OPAL a través de líneas de Smalltalk de su ambiente de programación, pero se debe reconocer que la aplicación del desarrollo de ambiente puede no ser el mismo para el ambiente en el cual fue finalizada la aplicación. Sin embargo, se esta aprobando los procedimientos proporcionados a las interfaces para C y Pascal.

ARQUITECTURA GemStone

La siguiente figura muestra las piezas más grandes del sistema GemStone. Stone y Gem que corresponden grandemente a la memoria de los objetos y a la máquina virtual de SmallTalk implementada. Stone provee administración secundaria del almacenamiento , control de concurrencia, autorización, transacciones, recuperación y soporte para accesos asociados. Stone también maneja los espacios de trabajo para sesiones activas. Stone usa puertas únicas llamadas object-oriented pointers (OOPs) puntos orientados a objetos para referirlos a objetos.



Stone usa una tabla de objetos para mapear un OOP a una ubicación física. Este nivel de indirección significa que los objetos pueden ser fácilmente movidos en la memoria. Mientras que la tabla de objetos puede potencialmente tener 2^{31} entradas, se espera que la porción para objetos actualmente en uso para varias sesiones sea tan pequeña como para entrar en la memoria principal. Stone es construida sobre un sistema VMS de archivo. El modelo de datos que Stone provee es tan simple que el modelo Gemstone es completo, adicionándole únicamente los operadores para actualización estructural y acceso. Y los objetos pueden ser almacenados separadamente desde sus subobjetos, pero los oops para los valores de un objeto de variables son agrupados juntos. Otros han considerado la descomposición de representaciones de objetos.

Gem se ubica en la cima de Stone y elabora los modelos de almacenamiento de Stone en el modelo completo GemStone.. Gem además añade las capacidades de compilación de los métodos OPAL en códigos de bytes, y ejecuta cada código, usando autentificación y sesiones de control. (Los códigos de bytes -bytes- OPAL son similares pero no idénticos a los usados en Smalltalk). Parte de la capa de Gem es la imagen virtual: la colección de clases de OPAL, métodos y objetos que están añadidos con cada sistema GemStone.
Comparando la jerarquía de la imagen actual de GemStone con la jerarquía de Smalltalk, se han removido clases para el acceso de archivos de comunicación, manipulación de las pantallas y en el medio ambiente de la programación. Las clases del archivo son innecesarias, de manera que hemos almacenado persistentemente todos los objetos de GemStone. La computación para la manipulación de las pantallas necesita ser cercana al usuario final, y necesita una ejecución rápida del código de bytes. GemStone es optimizada a través del mantenimiento de grandes números de objetos persistentes, más que por una ejecución rápida de códigos de bytes. Las clases de ambiente de programación son reemplazadas por una aplicación browser que corre en la cima de GemStone. Se han añadido clases y métodos para ser las funciones de administración de datos , de control de transacción, de conteo, de propiedad, autorización, replicación, uso de perfiles y creación de índices controlable dentro de OPAL.
El agente interface es un conjunto de rutinas para facilitar la comunicación de otros programas en otros lenguajes que corran en procesadores posiblemente remotos de GemStone. El agente interface actualmente soporta llamadas de C y Pascal corriendo en una IBM PC para sesión y control de transacción, enviando mensajes a objetos GemStone, ejecutando una secuencia OPAL, compilando métodos OPAL, y explicación de errores. En adición, el agente provee llamadas de "acceso estructural", las cuales permiten las siguientes funciones:

1. determinación del tamaño del objeto, clase e implementación;
2. inspección de clases definidas de objeto;
3. recuperando bytes o puntos de un objeto;
4. almacenando bytes o puntos en un objeto;
5. creando objetos.

La información pasa a través del Agente y Gem en la forma de bytes y GemStone como objeto de puntos.

USUARIOS MULTIPLES

Stone soporta múltiples usuarios actuales proporcionando a cada usuario un espacio de trabajo que contenga una copia de sombra de la tabla de objetos derivada de la más reciente tabla de objetos llamada la tabla compartida. Mientras la sesión modifica un objeto, una nueva copia del objeto es creada, y ubicada en la página que es inaccesible para otras sesiones.

Conceptualmente, la tabla objeto sombra para un espacio de trabajo es una copia completa de la versión de la tabla compartida cuando la sesión inicia. Actualmente no se hacen copias de todas a la misma vez. Se escogió un control optimista de esquema de concurrencia : En la cual cada acceso de conflicto es checado y aprobado en el tiempo, en lugar de prevenir a través de un sistema de candado. Para cada transacción Stone mantiene el canal en el cual el objeto de la transacción ha sido leído o escrito. Al tiempo de aprobación, Stone checa para cada conflicto leer-escribir y escribir-escribir con transacciones que han sido aprobadas desde el tiempo que la transacción iniciara. Si no hay conflictos la transacción es permitida para su aprobación. Para la aprobación, la tabla objeto sombra de una sesión es tratada como si fuera transparente en las entradas que no han sido modificadas, y es sobrecargada en la versión más reciente de la tabla compartida. Así, sólo las entradas en la tabla compartida para objetos que han sido copiados en la sesión son cambiados. En esta manera los cambios son sólo aprobados en la sesión y traídos de aquellos de otras transacciones que han sido aprobadas después de que se inicio la sesión de aprobación. Si la transacción actual confluctua con una transacción previamente aprobada (o que es abortada por solicitud), los cambios en esta tabla sombra son borrados, después de utilizar la tabla para recuperar páginas usadas para nuevas copias de objetos.
Este esquema optimista asegura que sólo las transacciones de lectura nunca confluctuen con otras transacciones. Tal transacción trae una copia consistente del estado de la base de datos, y no tiene cambios para hacer sobre la tabla compartida aprobada. Solo las transacciones que pueden escribir pueden confluctuar una con otra. Este esquema nunca tiene candados, es como una sesión, es decir no se tienen contenciones con otras sesiones antes del punto de aprobación. Sin embargo, es posible que una aplicación que escribe una gran porción en la base de datos puede fallar para aprobar cualquier transacción por un largo tiempo arbitrariamente. Mientras que el sombreado sólo tiene algunas malas presiones, parece una natural aproximación para nosotros, dado que nosotros tenemos una tabla de objetos. Y se evita algunas lecturas adicionales, haciendo aprobaciones y el aborto simple, y es un excelente candidato para escribir una vez en la memoria, dado que las páginas activas nunca son cambiadas de lugar.

CONSIDERACIONES DE EFICIENCIA

Un problema con la grabación de todos los objetos en una sesión de lectura o escritura es que la lista puede ser muy larga, y GemStone tardara mucho tiempo añadiendo entradas a tales listas. una optimización es tal que ciertas clases de objetos, tal como Smalltalk, Character, y boolean se conocen de que contienen sólo instancias que no pueden ser actualizadas. Así, tales objetos no necesitan ser grabados para control de concurrencia. Aún excluyendo estos objetos, los objetos simples son tan finos gradualmente para un control de concurrencia. De esta manera, se introduce la noción de segmentos, los cuales son agrupamientos lógicos de objetos que están en la unidad del control de concurrencia en

GemStone. Mucho muy similares a los segmentos de ADAPLEX LDM. Un segmento puede contener cualquier número de objetos. GemStone mantiene una lista de aquéllos segmentos leídos o escritos en una sesión, en lugar de todos los objetos. Así, en un nivel físico, las páginas respectivas a los límites de los segmentos. Por lo tanto la práctica de copiar todos los objetos en una página cuando una es cambiada no causa conflictos adicionales. Los segmentos son visibles desde OPAL a través de la clase Segmento. Los usuarios pueden controlar la ubicación de los objetos en los segmentos, agrupar los objetos tratando de evitar conflictos. Una aplicación que tenga un grupo de objetos privados, son todos aquellos objetos y no otros pueden ser ubicados juntos en un segmento. En el sistema de nivel, los sistemas de objetos que son compartidos por muchos usuarios, pero que casi nunca son actualizados (tal es una clase que describe los objetos de una clase sistema) y pueden ser ubicados en un segmento simple, de manera, que los accesos a ellos nunca causen conflicto.

TRANSACCIONES Y RECUPERACIÓN

Abortar una transacción significa tirar los objetos de la tabla sombreada, y aprobar significa reemplazar la tabla compartida con la copia sombreada. El sólo propósito que necesita más elaboración es la atómicidad, que cambia como una transacción cuando es hecha, similarmente, al mismo tiempo. Así los objetos de la tabla, son árboles, la atómicidad no es muy difícil de proveer. Cuando una tabla sombreada reemplaza a una tabla compartida, y la tabla sombreada difiere de la tabla que va a reemplazar, la nueva tabla puede reemplazar la vieja simplemente sobrescribir en la raíz de la tabla compartida de objetos dentro de la raíz de la tabla nueva. Actualmente, existe una "tabla de bases de datos" sobre la raíz de la tabla de objetos que ha sido sobrescrita. La raíz de la base de datos de referencia da alguna información junto con la tabla de objetos, de manera que se tiene una lista activa de transacciones. La rescritura es solo el único lugar donde cualquier parte de la copia compartida de la base de datos es sobrescrita.
La recuperación de una falla del procesador no requiere una gran cantidad de mecanismos adicionales sobre los cuales se tenga control de concurrencia. La unidad de recuperación es una transacción. Los cambios hechos por las transacciones aprobadas son mantenidas, los cambios no todavía aprobados son perdidas. Dado que la inversión compartida de la Base de Datos nunca es sobrescribir normalmente, casi nunca se necesita la manera de puntos para traer bases de datos en un estado de consistencia, dado que nunca dejan ese estado. La única parte de trampa es que el procesador rompe mientras escribe una nueva raíz de Base de Datos. Para manejar esta eventualidad, se mantienen dos copias de la raíz, una reside en un lugar conocido. Para establecer el estado de consistencia de la Base de Datos después del rompiendo, únicamente se checan aquellas dos páginas. Si son diferentes la copia puede ser determinada como incorruptible sobre la otra. El trabajo real de recuperación es una colección de basura: removiendo basura de las transacciones que no han sido aprobadas antes del rompimiento.

ACCESOS ASOCIATIVOS Y ESCRITURA

Nosotros brevemente cubrimos algunos de los objetivos de lenguajes y escrituras relacionados a los accesos asociados, a través del Indice de las estructuras y su mantenimiento. El lenguaje fundamental establecido es capaz de detectar las oportunidades para utilizar estructuras auxiliares de almacenamiento. En un lenguaje computacional completo como OPAL, no es necesario o deseable considerar estructuras auxiliares para cada manipulación de Base de Datos. Concebiblemente se pueden analizar todos los métodos de OPAL para detectar lugares donde accesos alternativos de transferencia pueden ser usados. Se puede observar que esta aproximación es muy compleja, y en lugar decimos que el programador debe poner banderas de oportunidad como estructuras auxiliares.
OPAL soporta el uso de Indices para acelerar la evaluación de expresiones de la forma :

aBag Select; aBlack

Este block tiene una variable y regresa a Booleano. El resultado de expresión es que un subconjunto de elementos de a Bag para la cuál a Block regresa a verdad true, y reensambla la selección relacional del operador en el modelo de datos Cypress. Este establecimiento es evaluable en OPAL sin Indices, pero a un costo de examinar cada elemento en Bag. Dado que un Block que puede contener expresiones arbitrarias de OPAL, los Indices no son útiles en la evacuación de cada expresión en un Block. Aquí, para el uso de estos Indices, únicamente se añaden trayectorias de sintaxis para el lenguaje OPAL. Para cualquier variable , se adiciona a la trayectoria compuesta una secuencia de piezas llamada uniones links, la cual específicamente es una subparte de un objeto. Por ejemplo, an Emp. Name.last puede accesar el último nombre de un objeto Empleado. Una consulta sobre sale porque la secuencia de los mensajes unarios no son suficientes para la misma cosa, tal como anEmp.Name.last. Las razones que se soportan a los accesos asociativos es al mismo tiempo de corrida sin permitir mensajes enviados, así el soporte puede venir de un nivel de Stone.
Un Block de Selección para accesos asociativos puede contener una conjunción de comparaciones de trayectoria, en donde una comparación de trayectoria es una expresión de la forma (path expression) (comparator) <literal> o <path expression> <comparator> <path expression>:

anEmp. emp. Names.last.= "Sanders"

anEmp. salary>an Emp. dept. manager. salary

El Indice usado es requerido utilizando conjuntos de brazos en lugar de brackets a lo largo del block en un mensaje de selección:

empSet select
{ : an Emp. emp Name. last Name= "Sanders" ·

Rather Than

```
emp Set Select:
 {:an Emp:an Emp. emp Name.last Name="Sanders"}
```

Las dos expresiones dadas tienen el mismo resultado, pero la primera pide a OPAL el uso de un índice que esta disponible, mientras el segundo siempre es evaluado tirando a través de emp Set. Si no tiene un índice apropiado, entonces la primera expresión podría ser evaluada sin el uso de enviar mensajes. De otra manera, la primera expresión evalúa usando el método como el segundo. Se encuentra una gran ayuda en probar accesos asociativos de proceso, que tienen una fuerza bruta de camino para evaluar selecciones de consultas, como una clase " bancos semánticos " para checar evaluaciones basadas en índices .
Otra decisión central en el diseño de accesos asociativos es cómo hacer el índice, las clases y las colecciones ?. Muchas aplicaciones pueden utilizar instancias de la misma clase, y almacenamiento en diferentes colecciones.
El indexado de clases significa que la aplicación no usa el índice; continúa arriba de las instancias que usan el índice. Así el índice de una clase amplia presenta problemas de autorización. Ningún otro usuario puede tener de lectura al conjunto de todos los objetos en la clase, y nadie es capaz de referir la creación del índice. Así, indexando una colección permite la posibilidad de que las instancias de subclases sean incluidas en una colección que es indexada. La indexación de una base de clases hace más fácil el seguimiento de seguir los cambios en el estado de un objeto que puede causar que el objeto sea posicionado diferentemente dentro de un índice. Se ha decidido minimizar el costo para la programación no usando índices que tengan prioridad más alta que los índices en colecciones y en otros sistemas han sido escogidos en base a la indexación de clases. Adicionalmente, si se indexan por clases la intersección de resultados de una vista superior con una colección puede consumir tiempo con respecto al tamaño de la colección. Nótese que una clase puede ser implementada para mantener una colección de todas las instancias que se desean, y que las colecciones pueden ser indexadas.
Los índices son creados y abandonados enviando mensajes a Bag ó Set objeto, dando la trayectoria al índice. Por ejemplo, si emp Set es un conjunto del objeto Employee, se puede requerir un índice sobre empName. Last. Hay dos clases de índices: identidad y equalidad. Un índice de identidad es el de algunos subobjetos de uno de los elementos, sin referencia a el estado del subobjeto. Un índice de igualdad supone visualizar en la base del valor o el estado interno del objeto, y el rango de búsqueda de éstos valores.

4.6 PICQUERY

Resumen.

Picquery ha sido diseñado de forma similar al Query by Example de IBM, como un lenguaje altamente conversacional y no procedural para un sistema de bases de datos gráfica, diseñado y desarrollado por UCLA. Picquery y el lenguaje de consultas Query by Example pudieran formar un conjunto de lenguajes a través de los cuales un usuario pudiera manejar un sistema de bases de datos relacional y al mismo tiempo manejar una base de datos gráfica. Esta interfaz forma parte de una arquitectura dirigida al manejo de bases de datos gráficas y no gráficas.

Introducción.

Muchos de los sistemas de manipulación de datos que han sido implementados para manejar información gráfica, tales como imágenes digitalizadas, dibujos, etc., fueron desarrollados para áreas de aplicación muy específicas, como por ejemplo, aplicaciones geográficas, exploraciones militares y aplicaciones en medicina. Se han desarrollado muy pocos sistemas de propósito general. Consecuentemente los lenguajes de acceso o consulta a este tipo de sistemas son también muy específicos.
Se han identificado y propuesto un conjunto de operaciones para la manipulación de datos que están soportados sobre un sistema de bases de datos gráfica y se ha desarrollado Picquery como un lenguaje de alto nivel para implementar esas operaciones.
Desde el punto de vista de arquitectura, este esfuerzo es parte de un proyecto de la UCLA, cuyo interés es el desarrollo de lenguajes y manipuladores para bases de datos gráficas y no gráficas, al igual que para bases de datos convencionales.

El modelo de datos y su manejo en Picture Database Management System (PICDMS).

PICDMS usa el esquema de representación en forma de rejilla para los datos de tipo gráfico. Posee la única estructura de datos de imágenes, lógica empilada dinámicamente, la cual consiste en almacenar en una misma rejilla todos los datos de un gráfico o imagen y aunque pareciera que estos valores se almacenan de forma convencional, la diferencia radica en que se hace de forma dinámica, no se prevé un tamaño para la base de datos y cada "récord" tiene el espacio que necesitan sus datos y aumentará éste si así lo requiriese la imagen. Esto permite flexibilidad y eficiencia en la adición y borrado de datos, que es un requerimiento esencial para los manejadores de bases de datos. Una nueva imagen es añadida como un atributo en un récord y no como un nuevo récord en la estructura convencionalmente estática ya conocida, ya que no soporta este esquema dinámico.

El acceso a los datos se hace a través de un algoritmo que muestra una ventana rectangular que contiene las celdas con la pila en la que están almacenados los datos de la imagen en el orden en que fueron capturadas. El lenguaje de manipulación de datos de PICDMS es un lenguaje procedural, con la siguiente sintaxis:

<Comando> :: <Nombre de comando>
(<conjunto de variables>)
<conjunto de campos>,
FOR <condición>;

El nombre del comando especifica la operación que se desea realizar. El conjunto de variables define las variables que se usarán en la ejecución de la operación. El conjunto de campos son aquellos sobre los que se ejecutará la operación y la condición tiene el objetivo de filtrar aquéllos valores de campos que se deseen. Las operaciones principales y sus correspondientes comandos son: COMPUTE, LIST, DISTANCE, ADD, REPLACE, DELETE and PRINT. Con estos comandos se pueden ejecutar casi todas las operaciones de manipulación de los datos de una base de datos gráfica.
Las operaciones fundamentales de los lenguajes de consulta convencionales no son manejadas por este lenguaje y las que se pueden realizar ocupan mucho tiempo de procesamiento, dado por las características de los datos que contiene la base de datos.
Picquery tiene dos formas de funcionar:
1.- como lenguaje no procedural puro, que es bueno emplear para el acceso en línea. Sin embargo cuando el número de operaciones es muy grande se vuelve muy engorroso,
2.- formato tabular de consultas, que es una forma muy fácil de escribir una solicitud de información a la base de datos, pues el lenguaje muestra un esquema preestablecido de consulta para cada operación y el usuario sólo debe llenar en la tabla el valor de los datos que se piden.
Muchas áreas de tratados y revistas especializadas en sistemas de administración de datos se examinaron para proporcionar el común denominador del acceso ilustrado de datos y la manipulación de las operaciones requeridas.
Tres grandes áreas de aplicación ilustran las operaciones comunes que podrían estar disponibles en un PDBMS generalizado : Geográfica industrial y medica.
Un PDBMS para sistemas de información gráfica ilustrada necesita proporcionar las siguientes operaciones en el análisis de la imagen de datos: La capacidad de observar imágenes con diferentes niveles de detalle (zooming), rotación de imágenes por diferentes ángulos, tabulación cruzada de datos, identificación de los objetos más cercanos, operaciones de estadísticas, como medidas de promedios, operaciones geométricas o espaciales, enclaustrado y clasificación de puntos, detección de umbrales
El PDBMS industrial en contraste con el geográfico tiende a trabajar con volúmenes mas pequeños de datos ilustrados y estos sistemas pueden efectuar

cálculos complejos sobre las bases de datos de imagen. Operaciones tales como rotación de imágenes, acercamiento, detección de filos, ajuste de plantillas, medición de texturas, cálculos geométricos y medidas estadísticas, son relevantes para estos sistemas.
Los PDBMS médicos son utilizados para el almacenamiento de imágenes de rayos X , también trabajan con grandes cantidades de datos. Las operaciones importantes para estos sistemas incluyen el mejoramiento de rotación de imágenes, extracción de contornos y segmentación para diagnostico automático y semiautomático.
En general el PDBMS debe proporcionar muchas de estas capacidades de manejo de datos (compartidas a través de varias áreas de tratado) como sea posible. Las diferentes capacidades de manejo de datos que deberán ser proporcionadas por un PDBMS pueden ser clasificadas en seis categorías .

Principales operaciones del manejador de base de datos gráfica.

1.- Operaciones de manipulación de imágenes.
Estas operaciones permiten ofrecer perspectivas diferentes de una misma imagen.

- Trasladar o panear (shifting operations).
- Rotar (Rotation).
- Zoom (Zooming operations).
- Superponer (Superimposing operations).
- Transformación de colores (Color transformations operations).
- Proyección (Projections operations).

2- Operaciones de reconocimiento de patrones.
Estas operaciones permiten reconocer y dibujar patrones establecidos o buscar aquellas imágenes que en la base de datos gráfico coinciden con dichos patrones.

- Detección de filos; Para detectar filos mediante cambios en la intensidad de la luz a través de la figura.
- Umbral; para construir una imagen binaria que sea blanca en las regiones con intensidad de luz menor que un límite de umbral y negra en el resto.
- Dibujado de contornos; para dibujar líneas de contorno en la figura asociada con el mismo valor de atributo.
- Recuperación de similaridad (ajuste de plantillas); para identificar o recuperar objetos dibujados que sean similares a otros de otra figura usando una cierta medida de similaridad o bien que cumplan con ciertos patrones de plantilla. La recuperación por similaridad puede ser hecha sobre la base de tamaño, perfil, textura, etc.
- Establecimiento de la frontera o perímetro de una región.
- Medición de texturas para medir o cuantificar la textura de una imagen .

- Agrupamiento (clasificación de puntos); para agrupar objetos o puntos que están cercanos en una figura.
- Segmentación; para dividir una figura sobre la base de un mismo criterio.
- Interpolación; para interpolar valores de puntos dispersos de una función en particular.
- Vecino más cercano que identifique y recupera el objeto más cercano (de un tipo particular).

3.- Operaciones geométricas o espaciales.

- Determinación de distancias punto a punto, punto a línea, línea a línea, línea a región y región a región.
- Determinación de longitudes.
- Determinación de áreas.
- Búsqueda de líneas iguales en diferentes imágenes.
- Búsqueda de regiones iguales en diferentes imágenes.
- Determinación de intersecciones.
- Unión de dos regiones.
- Determinación de diferencias.

4.- Operaciones funcionales.

- Cálculo de máximos, mínimos, valores totales, contadores.
- Funciones estadísticas (promedios, desviaciones estándar, histogramas, intervalos, referencias cruzadas, etc.)

5.- Funciones definidas por el usuario (UDFs) e interfaz del lenguaje de programación.

6.- Operaciones de entrada y salida.

- Listar o imprimir la información de una imagen.
- Grabar una imagen o partes de ella en algún dispositivo de almacenamiento.
- Colorear imágenes.
- Cambiar o actualizar los valores de una imagen.
- Almacenar nuevos datos relacionados con alguna imagen.

El lenguaje de consultas Picquery.

El lenguaje de consultas Picquery puede operar en bases de datos, gráficas completas o en parte de ellas. Una imagen es identificada de manera particular y un objeto de ella es por ejemplo, el nombre, los valores que describen una línea,

una región o un punto. Al hacer una consulta se hace referencia a uno de estos objetos o a la imagen completa y si al realizar la consulta Picquery no encuentra estos nombres ofrece como respuesta la base de datos completa. Asimismo hay operaciones que sólo pueden ejecutarse sobre la imagen completa y otras sólo sobre partes de ella.
Estas funciones estarán disponibles para aquellos usuarios que desean implementar nuevas funciones y construir consultas con ellas.
A continuación se describirán ejemplos de operaciones que se pueden realizar con Picquery.

Ejemplo 1. Rotación.
El usuario deberá escribir la palabra ROTATE con lo que se desplegará la siguiente tabla:

ROTATE

Picture	Object Name	Angle of rotation from vertical axis
PIC	STRIPE	-10

Al ejecutarse esta operación se rotará el objeto STRIPE de la imagen PIC diez grados en sentido contrario a las manecillas del reloj.

Ejemplo 2. Transformación de color.
El usuario deberá escribir la palabra COLOR TRANSFORMATION con lo que se desplegará la siguiente tabla:

COLOR TRANSFORMATION

Picture	Object Name	Color	Section conditions			
			Variable Group	Relation Operator	Value Operator	Logical
PIC	UTAH	Blue	Elevation 1	.gt.	1000	.and.
			Elevation 1	.le.	2000	
		Red	Elevation 2	.gt.	2000	

Al ejecutarse esta operación en la imagen se colorearán de azul las elevaciones mayores que 1000 pies y menores que 2000 pies y de rojo las mayores que 2000.

Ejemplo 3. Longitud de una línea.
El usuario deberá escribir las palabras LENGTH OF A LINE con lo que se desplegará la siguiente tabla:

LENGTH

Picture	Line Name	Length
PIC	NILE	P.

Al ejecutarse esta operación se devuelve la longitud del objeto NILE en la imagen PIC.

Conclusión.

Se ha presentado la motivación y justificación para el desarrollo de un lenguaje altamente interactivo para el incremento y desafío de la administración generalizada de datos gráficos. PICQUERY y un lenguaje relacional tipo QBE formarían el lenguaje mediante el cual el usuario puede accesar bases de datos relacionales y al mismo tiempo administrar base de datos por PICDMS (u otros PDBMSs fuertes). También se ha presentado un conjunto racionalmente comprensible de accesos a datos gráficos y operaciones de manejo requeridas por un PDBMS. El lenguaje PICQUERY diseñado provee el soporte para la gran mayoría de estas operaciones utilizando pocos comandos de entrada. Aparentemente los PDBMSs convencionales (basados en lenguajes relacionales, CODACYL, DBMS jerárquico) no proporciona el conjunto adecuado de operaciones fundamentales para soportar adecuadamente la mayoría de las operaciones gráficas.
Un subconjunto significante de PICQUERY ha sido ilustrado a través de varios ejemplos, no obstante que la definición completa del lenguaje no ha sido incluida. Es claro que la aproximación PICQUERY es un lenguaje finalmente abierto el cual puede ser extendido para acomodar funciones nuevas o adicionales que pueden ser de interés particular para áreas específicas. De hecho la arquitectura de los PICDMS, de los PICDMS DML orientados a procedimientos y de PICQUERY es tal que proveen una estructura fundamental sobre la cual posteriores operaciones de lenguaje y necesidades de manejo de datos pueden ser construidas fácilmente.

5. Conclusiones

La necesidad de soportar la funcionalidad orientada a objetos en una base de datos es guiada también por las exigencias de una determinada aplicación. Las bases de datos orientadas a objetos toman ventaja en las aplicaciones en las que las relaciones entre elementos de la base de datos llevan la información clave. Las bases de datos relacionales salen beneficiadas cuando los valores de los elementos de la base de datos llevan la información clave. Es decir, los modelos orientados a objetos capturan la estructura de los datos; los modelos relacionales organizan los datos en si. Si un registro puede ser comprendido aisladamente, entonces la base de datos relacional es adecuada. Si un registro tiene sentido solamente en el contexto de otros registros, entonces es más apropiada una base de datos orientada a objetos.

Las aplicaciones técnicas y de ingeniería fueron las primeras en requerir bases de datos que manipularan tipos de datos complejos y capturaran la estructura de los datos.

El alto nivel de productividad alcanzado en la investigación y el desarrollo durante los últimos diez años se ha debido principalmente al hecho de que los investigadores y diseñadores en SABDOO han reutilizado y adaptado la tecnología y las experiencias acumuladas en el desarrollo de sistemas relacionales en los años 70. La fase exploratoria y de experimentación para la tecnología en SABDOO esta en vías de terminar y esta haciendo raíces para lanzarse a los ambientes productivos y llenar las necesidades que los sistemas tradicionales de Bases de Datos no han sido capaces de satisfacer. Además, la transición de la fase actual a la siguiente tomará solo algunos años, requiriendo mayor investigación y consolidación. La investigación estará dirigida hacia el establecimiento de puentes entre los SABD tradicionales y la tecnología SABDOO, y los mejores resultados de los esfuerzos de investigación y desarrollo realizados hasta la fecha, no solo en el área de los SABDOO sino también de las BD extensibles, las BD distribuidas y las interfaces de usuario se consolidaran para construir la siguiente generación de SABDOO para ambientes de desarrollo productivos.

La actualización de SQL en Mayo de 1993 discutida en diferentes aspectos del trabajo en progreso para la siguiente versión de el SQL standard, el cual ha sido actualmente desarrollado en ANSI y ISO como "SQL3".

La más reciente edición de SQL3 (ANSI X3H2-93-091 y ISSO DBLYORK-003) es poco menor que 1,000 páginas. La reciente publicación de SQL-92 standard es aproximadamente de 600 páginas, así que esto es fácil para resumir que SQL3 esta ya casi 50% más grande y esto no esta todavía terminado. Alguien quien actualmente trabaja con el SQL-92 standard esta siendo intimidado por éste hecho; esto es totalmente voluminoso.

Con esta nueva metodología ya no es necesario para el diseñador de sistemas convertir el dominio del problema en estructuras de datos y control predefinidas, presentes en el lenguaje de implementación. En vez de ello, el diseñador puede

crear sus propios tipos de datos abstractos y abstracciones funcionales y transformar el dominio del mundo real en estas abstracciones creadas por el programador. Esta transformación, incidentalmente, puede ser mucho más natural debido al virtualmente ilimitado rango de tipos abstractos que pueden ser inventados por el diseñador. Además, el diseño de software se separa de los detalles de representación de los objetos de datos usados en el sistema. Estos detalles de representación pueden cambiar muchas veces, sin que se produzcan efectos inducidos en el sistema de software global.
La naturaleza única del diseño orientado al objeto está ligada a su habilidad para construir, basándose en tres conceptos importantes del diseño de software: abstracción ocultamiento de la información y la modularidad. Todos lo métodos de diseño buscan la creación de software que exhiba estas características fundamentales, pero solo el diseño orientado a objetos (DOO) da un mecanismo que facilita al diseñador adquirir los tres sin complejidad o compromiso.
Finalmente, el Diseño OO ha evolucionado como resultado de una nueva clase de Bases de Datos Orientadas a Objetos, tales como GemStone, Vbase, Iris, etc.
Las dos tendencias básicas en BDOO son:

- Agregar capacidades a los paquetes existentes (extensión del Modelo Relacional)
- Incorporar persistencia a los lenguajes Orientados a Objetos (como C++ o Pascal).

Aparentemente las dos tendencias evolucionarán favorablemente y coexistirán incluso puede pensarse que tendrán capacidad de interconectarse para importar y exportar datos.

6. APENDICES

6.1 BIBLIOGRAFIA

OBJECT- ORIENTED
PROGRAMATING: AN INTRODUCTION
GREG VOSS
Osborne Mc Graw-Hill, 1991

TURBO PASCAL 7
Manual De Referencia
Stephen O' Brien
Osborne /Mc Graw - Hill, 1993

Curso de Programación C ++
Programación Orientdada a Objetos
Fco. Javier Ceballos
Addison-Wesley IBEROAMERICANA/ra-ma, 1991

Aplique S Q L
James R. Groff
Paul N. Weinberg
Osborne Mc Graw-Hill, 1992

Aplique C ++
Bruce Eckel. Incluye C ++ Versión 2.0
Osborne Mc Graw - Hill, 1991

Object - Oriented Concepts,
Databases, and Applications
Editado por Wonkim y Press/Frontier Series, 1989
Frederick H. Lochvsky

C ++ A su Alcance
Un enfoque Orientado A Objetos
Luis Joyanes Aguilar
Mc Graw - Hill, 1994

Turbo Pascal 6.0
Luis Joyanes Aguilar
Serie Mc Graw - Hill De Informática, 1993

Borland C++ 4
Object - Oriented
Programming
Third Edition
The Power Programmer's Guide To
Object - Oriented Development
Faison/SAMS PUBLISHING, 1994

Programación Orientada a Objetos en C++
Miguel Katrib Mora
INFOSYS, 1994

Introducción a los Sistemas de
Bases de Datos Volúmen 1
C. J. Date
ADDISON - WESLEY IBEROAMERICANA, 1993

OBJECT-ORIENTED DATABASES
WITH APPLICATIONS
TO CASE, NETWORKS, AND ULSI CAD
Rajiv Gupta
Ellis Horowits
Prentice - hall, 1991

Object-Oriented Database System
Concepts and Architectures
Elisan Bertino, Lorenzo Martino
Addison - Wesley Publishing Company, 1993

Object Data Management
Object-Oriented and Extended Relational
Database Systems
Addison - Wesley Publishing Company, 1994

Object Databases
THE ESSENTIALS
Mary E. S. Loomis
Addison - Wesley Publishing Company, 1995

Research Foundations in Object-Oriented
and Semantic Database Systems
Alfonso F. Cárdenas, Dennis McLeod
Prentice-Hall, 1990

6.2 LISTA DE EMPRESAS

GEMSTONE OBJECT-ORIENTED DBMS 4.0
Sistema Manejador de Bases de Datos Orientado a Objetos
GemStone 4.0

Un ODBMS de gran calidad diseñado para empleo en grandes aplicaciones comerciales. Emplea una arquitectura activa de bases de datos que permite la ejecución de datos y métodos directamente en el servidor de la base de datos. Permite aplicaciones escritas en Smalltalk, C++ y C para accesar y modificar los objetos almacenados en la base de datos actualmente. Sus caracterlsticas incluyen mecanismos avanzados de seguridad, integridad referencial, control de concurrencia, bajar información (backup) y recuperación rápida. Soporta un medio ambiente con gran disponibilidad que requieren operaciones de 24 horas con grandes capacidades como la modificación de esquemas dinámicos y la colección en llnea de desperdicios. Soporta ambientes de plataforma mezclados.

HyBase
Answer Software Corp, Cupertino, CA.
Un servidor de Bases de Datos estructurable para Macintosh que permite a los usuarios crear tablas usando operadores SQL, asl como Select. Incluye clases de objetos definidas por el usuario tan bien como gráficas y sonidos. Proporciona dentro de su construcción un escritor de reportes para control de tipo de letra y posicionamlento, y una interface programable para soportar la generación de reportes complejos. Pueden manejarse solicitudes para clientes múltiples y asegurar objetos tales como registros o campos para prevenir que otros clientes accesen esos datos que han sido actualizados por otro usuario. Son compatibles con AppleTalk.

IDB OBJECT Database 2.5
Persistent Data System Inc.,
Pittsburgh, PA.
Una base de datos distribuida de objetos que permite al desarrollador definir y manipular objetos de tamaño y complejidad arbitraria. Organizados en una clase de jerarqula de herencia múltiple, los objetos deben de tener ambos datos (atributos) y el comportamiento asociado (operaciones).
Las clases de objetos son definidas interactivamente usando el esquema diseñado IDB. Las operaciones de objetos son programadas en C. El tiempo de corrida del sistema IDB soporta el polimorfismo, uniones dinámicas, alocación de almacenamiento rápido y recolección de desperdicios, transacciones largas y anidadas, versionamiento y el manejo por excepción. Las herramientas IDB ayudan en el prototipo, en la evaluación del programa y en la búsqueda. Los browser IDB proporcionan accesos interactivos para capacidades IDB. Las aplicaciones pueden correrse aisladamente o en forma distribuida, y pueden ser configuradas con o sin el browser y el administrador de display portátil. Las

aplicaciones son portátiles y los datos pueden ser compartidos a través de estructuras heterogéneas de todas las plataformas soportadas, incluyendo Macintosh, Windows, NeXT, Sun, IBM RS/6000, y HP.

Matisse
ADB Inc., Cambridge, MA.

Un DBMS que combina las capacidades del modelo orientado a objetos con una misión crítica de tecnología transaccional. Sus características incluyen tolerancia a las fallas, consistencia de datos e integridad, el acceso al OLTP, con alta resolución, la administración histórica de objetos, y el modelamiento inteligente de restricciones.
Proporciona además almacenamiento de datos común y concurrente, y la retribución de capacidades para aplicaciones múltiples. Soporta arquitecturas heterogéneas cliente/servidor. Corre en Sun SPARCstation, VAX-VMS; HP9000, WindowsNT Client, Windows 3.1, y Macintosh, Sistema introductorio Matisse Jr.

ObjetivitiTy/DB 3.5
ObjetiviTy inc., Mountain View, CA.

Soporta tanto arquitecturas de servidores multienlazados así como arquitecturas distribuidas escalables objetivas/DBS. Sus capacidades incluyen la integridad referencial, la tolerancia, los errores y el soporte para el empleo de aplicaciones de misiones críticas. Soporta el desarrollo de lenguajes en interfaces para C++ y ParcPlace Smaltalk, tan bien como ANSI, standard SQL, con soporte ODBC para aplicaciones integrativas en ambientes existentes de bases de datos.
Sus plataformas son: DEC VAX y Alpha con OSF/1, etc.

Omniscience ORDBMS
Omnimiscience Object Technology Inc.,
Santa Clara, CA.

Ofrece varias aplicaciones de interfaces en ambas relaciones y paradigmas de objetos orientados. Un desarrollador puede escribir una aplicación ya sea en SQL o en un lenguaje de programación orientado a objetos como en C++. Soporta objetos interoperacionales a través de lenguajes y modelos de datos. Un DBMS escalable con un pequeño impresor footprint de 1MB de RAM, el cuál puede operar confortablemente en máquinas con configuraciones pequeñas tales como computadoras notebook corriendo Windows. En grandes estaciones de trabajo o servicio, los pequeños footprint incrementan la utilización de la memoria virtual, generando menos fallas de página, e incrementando igualmente la ejecución del sistema. Hace uso extensivo de caches dinámicos, tales como buffers de página y objeto cache. Confirma estándares de SQL, incluyendo el ANSI SQL-92 y ODDC 2.0, entre otros. Es disponible para Windows 3.1, Windows NT, Sun Solaris y Macintosh.

Ontos DB 3.0
Ontos Inc., Burlington, MA.

Un objeto componente DBMS que proporciona aperturas, flexibilidad y extensibilidad en el desarrollo de escalables, distribución a las aplicaciones de objetos orientados. El barco con bandera del producto de Ontos y de Arquitectura con información virtual, una estructura de componentes de objetos para accesar información y distribución. Disponible para desarrolladores de Ontos y para uso selectivo independientemente de core extendida de la base de datos y sus características para alcanzar las necesidades de sus aplicaciones, negocios o empresas. Esta escrito en soportes C++, y en una interface de objeto SQL, es disponible para la mayoría de las plataformas Unix.

Phyla
Mainstay, Camarillo, CA.

Un usuario final de base de datos orientada a objetos para PC's que permite a los usuarios recolectar, almacenar, analizar y presentar información compleja. Los usuarios pueden relacionar información de una gran variedad de fuentes, diseño y otras en formas de bases de datos comunes, y crear búsquedas de consultas complejas, sin ninguna programación. Sus características incluyen estructuras naturales intuitivas de objetos y clases, y una interface drag-and-drop.
Usa diagramas que representan clases de objetos y sus relaciones. Los objetos son relacionados por rastreo y tirados de una clase a otra. Los usuarios trabajan ya sea en una sobrevista fuera de línea que muestra las clases y sus relaciones o bien en formas comúnmente diseñadas que automáticamente permiten cambios conceptuales. Corre bajo sistemas macintosh 6.0.3 o posteriores.

POET 3.0
POET Software Corp., San Mateo, CA.

Un ODBMS escalable portátil y heterogéneo para C++ para un rango de aplicaciones que corren en plataformas de esta computadoras notebook hasta grupos de trabajo de sistemas cliente/servidor. Adiciona características de las bases de datos a las características de C++ mientras soporta herencia de C++, polimorfismo, puntos de referencia, objetos encapsulados. No necesita escribir códigos de traducciones de objetos en tablas. Las características de las bases de datos incluyen consultas de objetos, clasificaciones, indexación, transacción bloqueo de objetos, versionamiento de clases, contenido de clases, cadenas de longitud variable y tipos de BLOB. Nuevas características incluyen soporte de OLE 2.0, ODBC, y ODMG-93; esquema de versión check-in/check-out; y lenguaje de consultas de objetos(OQL). Es disponible para Windows 3.1, Macintosh, Windows para trabajo en grupos, Windows NT, Novel, OS/2; etc.

Raima Object Manager
Raima Corp., Issaquash, WA.

Interface para programación C++ para el Administrador Raima de Bases de Datos y Velocis. Una librería de clases C++ que permite a los desarrolladores adicionar persistencia a los objetos que son usados en programación y en aplicación con una máquina de base de datos Raima.

Total ORDB
Cincom Systems Inc., Cincinaty, OH.

Un componente de la base de datos Objeto-Relacional de administrador de tecnología de Total Frame Work, Cincoms integrada en un conjunto objeto de software tecnológico de productos y servicios.
Un sistema Manejador de Bases de Datos objeto-ralacional diseñado para un cliente/servidor, sistemas abiertos para ambientes de computadoras. Combina todas las capacidades de prerelacional, relacional y tecnología orientada a objetos en bases de datos en un simple sistema unificado. Sus características incluyen un modelo objeto-relacional de datos, ANSI SQL con extensiones orientadas a objetos y estructuras de integración de datos de multimedia.

UniSQL/M Multidatabase
System 3.0
UniSQL Inc., Austin, TX.

Un sistema multibase de datos que permite el desarrollo de la aplicación cliente/servidor utilizando vistas simples de sistemas de bases de datos múltiples heterogéneas. Ofrece incrementar la productividad para el desarrollo de aplicaciones quienes desarrollan aplicaciones de misión crítica que requieren de lecturas simultáneas y actualizaciones para una colección de base de datos de multivendedores, incluyendo Ingress, Oracle, Sybase, y UniSQL/X. Condensa en interfaces de bases de datos múltiples en un lenguaje que simplifica la administración de las diferencias esquemáticas entre bases de datos múltiples cuando se actualizan simultáneamente ellos dentro de una transacción simple. Los desarrolladores pueden usar lenguajes de programación orientados a objetos, tales como C++, Smalltalk, y Object SQL para el desarrollo de aplicaciones con DBMSs que ya existen. Maneja colecciones de bases de datos de multivendedores, mientras que mantiene la autonomía local de las bases de datos. Proporciona drivers para DBMSs, incluyendo Adabas, DB2, CAIDMS, IMS, CA-Ingres, Oracle, etc.

UniSQL/X Database Management
System 3.0
UniSQL Inc., Austin, TX

Un sistema unificado de bases de datos objeto-relacional que permite a los usuarios mover un diagrama orientado a objetos, mientras preserva las inversiones existentes en aplicaciones basadas en SQL y el desarrollo de habilidades. Los desarrolladores pueden manejar base de datos relacionales, complejas, datos objeto, así como datos multimedia sofisticados con el mismo nivel de escabilidad, utilidad, accesibilidad y seguridad. Soporta un amplio rango de interfaces industriales estándares y herramientas, incluyendo C++, Smalltalk, C, ODBC, SAG CLI, ANSI SQL, Object SQL, Microsoft Access. Permite a los usuarios leer y actualizar una colección de bases de datos de multivendedores, incluyendo Adabas, DB2, CA-IDMS, IMS, CA-Ingres, Oracle, etc.
Los usuarios pueden también proyectar soportes para nociones orientadas a objetos (métodos, composición y herencia) en almacenamientos existentes de datos. Corre en Sun SPARCstation o SPARC compatible, HP9000, IBM RS/6000, etc.

Versant ODBMS 4.0
Versant Object Technology Corp.,
Menlo Park, CA.

Un ODBMS para multiusuarios, en ambientes distribuidos. Orientado hacia el manejo de estructuras con aplicaciones. Ofrece soporte para C++ y Smalltalk. Sus características incluyen arquitecturas basadas en objetos para usos de objetos lógicos IDS que garantizan la distribución y facilidad del desarrollo. Soporta independencia del compilador, permitiendo a los objetos para ser compartidos entre muchas plataformas. Diseñado para soportar un medio ambiente de alta disponibilidad de 24*7, con un respaldo en línea, evolución de esquemas dinámicos, y las características dinámicas del administrador de espacio.

6.3 GLOSARIO DE TERMINOS

Atributo.- Un atributo es la abstracción de una característica de un objeto, y éstos pueden ser:

- Completos.- que contengan toda la información pertinente al objeto descrito.
- Exclusivos.- que cada atributo refiera un aspecto diferente.
- Mutuamente independiente.- que tomen su valor independientes uno de otro.

BDOO.- Ofrecen parte de la misma funcionalidad que los lenguajes orientados a objetos. Permiten la encapsulación, activan métodos mediante mensajes a los objetos. Además de las bases de datos orientadas a objetos ofrecen a las bases de datos tradicionales la funcionalidad de la que carecen los lenguajes O.O. como persistencia y participación.

Clase.- Una clase es una familia de objetos de un tipo en particular. Los objetos que pertenecen a una clase son llamados instancias, por ello una clase puede tener una o varias instancias, pero una instancia tiene solo una clase, esto se conoce como herencia simple.

Componentes privados.- Son las partes a las que se puede tener acceso sólo por los métodos de la clase misma.

Componentes protegidos.- Son los datos y métodos que una clase padre puede heredar a sus descendientes como únicos herederos que los pueden manipular.

Componentes públicos.- Indica la verdadera interfaz; los elementos que son accesibles desde fuera de la clase.

Encapsulamiento.- Es la combinación de datos y procedimientos juntos en una declaración de objetos. Es la manera de ocultar los detalles de la representación interna de un objeto presentando solamente la interfaz disponible al usuario.

Excepciones.- Son condiciones no usuales en un programa. En general, son anomalías de programas en tiempo de ejecución tales como división por cero, desbordamiento, o de rangos de arrays y agotamiento de almacenamiento de memoria libre.

Herencia.- La herencia es una de las propiedades más notables de la programación orientada a objetos. Cuando un objeto se deriva de otro objeto, se obtienen todas las propiedades de ese otro.

Interface de usuario O.O..- Son únicamente las pantallas que se pueden ir creando para que el usuario interactúe con la Base de Datos, independientemente que la misma sea una Base jerárquica o relacional también orientada a objetos.

Mensaje.- Es el proceso de presentar a un objeto una solicitud para realizar una acción específica.

Métodos.- Método es tan solo otro nombre para PROCEDIMIENTO. Algunas veces a los métodos se les denomina funciones miembro porque son miembro de un objeto, reciben toda la información necesaria para llevar a cabo sus responsabilidades dadas y nada mas.

Objeto.- Un objeto es una abstracción de un conjunto de cosas del mundo real de forma que:

- Todas las cosas del mundo real del conjunto - las instancias - tengan las mismas características.
- Todas las instancias estén sujetas a las mismas reglas.

Persistencia.- Es cuando un objeto continúa existiendo después de que la aplicación del programa ha finalizado. Es la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (por ejemplo, el objeto continúa existiendo después de que su creador cesa de existir).

Polimorfismo.- Permite la manipulación de objetos de clases distintas como si fueran de la misma clase, con lo cual es posible definir interfaces uniformes para diferentes tipos de objetos.

SQL3 (Nuevo Standard).- Que se encuentra en su etapa final de revisión y que será aprobado y publicado en 1977. El cuál incluye entre otras cosas un lenguaje procedural propio y una extensión al modelo relacional para incluir objetos.

Tipos abstractos.- Es un tipo de dato definido por el usuario y que representa la abstracción de algún objeto; por tanto se ajusta al problema que se está resolviendo y puede ser manipulado como los tipos internos del lenguaje.

Templates.- El concepto de template expresa la genericidad y define clases parametrizadas, además de un nombre de tipo, el argumento de un template puede ser una cadena de caracteres, un nombre de una función o una expresión constante.