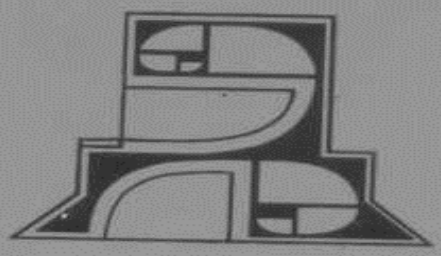
CUARTA ESCUELA INTERNACIONAL DE INVIERNO EN TEMAS SELECTOS DE LA COMPUTACION

de 22 al 26 de noviembre de 1993.
Mérida, Yucatán, México

Adolfo Guzmán Arenas

Diseño y construcción de aplicaciones con objetos



DISEÑO Y CONSTRUCCION DE APLICACIONES CON OBJETOS

ADOLFO GUZMAN ARENAS

2. Programación con objetos

Introducción

En vez de mantener por separado los datos de los programas, la programación con objetos junta los programas (que ahora se llaman métodos) con las estructuras de datos, para formar los *objetos*. Las estructuras de datos pueden compararse a un supermercado donde cada cliente va y toma las cosas que desea; cada programa accesa las estructuras de datos a su arbitrio. En vez de esto, a las propiedades (que ahora se llaman ranuras) de los objetos solo se puede tener acceso a través de los métodos de tal objeto. Por ejemplo, para obtener el valor de la edad de un futbolista, le envío un mensaje (una petición) pertinente: el nombre del método. El objeto nos regresa la edad. Esto muestra otra propiedad interesante: el encapsulamiento. ¿Cómo obtuvo el objeto su edad? No lo sabemos. Probablemente lo tomó de alguna ranura que él tenía. O tal vez lo calculó. "Los objetos solo nos dan lo que quieren." El acceso a las ranuras es más ordenado. Compárese con una tienda donde los artículos los surte una persona a través de una ventanilla.

Otra de las ventajas es la organización en clases. Una clase es un conjunto. También es un machote o cartabón con el cual se fabrican objetos específicos. A su vez, las clases tienen subclases (subconjuntos) y superclases (super conjuntos). De esta manera, muchos métodos no necesitan volver a definirse para una clase dada, sino se "toman" (se heredan, es el término usado) de sus clases superiores. Sin embargo, una clase específica puede derogar o modificar la definición de un método (o de una ranura) heredado de una superclase, y conservar el mismo nombre. Existe el polimorfismo: una misma función genérica se convierte en (o se comporta como) una de varias funciones específicas, de acuerdo con el tipo de argumentos: por ejemplo, suma se convierte en suma_entero, suma_real, o suma_complejo.

Descripción global

Se cubren primero definiciones y conceptos relevantes en el mundo de los objetos. Luego, en particular, se cubre un lenguaje de programación con objetos: C++.

En una segunda parte de este capítulo se describe el sistema CYC, que es una base de muchos conocimientos sobre sentido común o "conocimiento común." El proyecto CYC, que durante 10 años (desde 1984) se llevará a cabo en el laboratorio de investigación MCC de Austin, Texas, EEUU, es importante porque trata de dotar a los sistemas expertos y otros programas con "sentido común" o conocimiento superficial de muchas cosas, necesario para no cometer errores cuando al sistema experto se le hacen preguntas o solicitan operaciones fuera de su área de experiencia.

Artículos relevantes

[25], [42], [43], [45], [46], [59]. La lista de referencias se encuentra en la sección "Contenido."

2.1 Conceptos básicos

Uno de los primeros proponentes del uso de objetos fue Alan Kay, creador del lenguaje SmallTalk. Los objetos se basan en varias propiedades fundamentales: abstracción, encapsulamiento, clases, herencia a través de ellas; polimorfismo.

2.1.1 Abstracción

El diseñador de un programa no debe preocuparse muy temprano por la implementación concreta de sus estructuras de datos: arreglos, listas ligadas, etc. En vez de esto, se prefiere que se concentre en entender qué cosas (pronto serán llamadas objetos) el programa maneja, y las características que tienen en la vida real, desde el punto de vista de la aplicación.

Por ejemplo, desde el punto de vista de un programa de renta de videocintas a particulares (a familias, no a negocios), un "cliente" es un concepto importante que el programa va a manejar. Sus características son: nombre, domicilio teléfono, cintas que rentó, fecha de arrendamiento, saldo, tipos principales de cintas que renta, número total de cintas rentadas, en lo que va del año. En cambio, son de menor importancia (y pueden dejarse para el final, o incluso, idealmente, dejar que el compilador o herramienta de programación se encargue de ellas): un cliente es un arreglo, o una pila; comienzan sus elementos a contarse en 0, o en 1; el nombre del cliente tiene 30 caracteres, etc.

2.1.2. Objetos, clases, individuos, herencia

Las clases pueden verse como moldes o esqueletos para formar objetos. Los objetos son también llamados individuos, son instancias de una clase. Por ejemplo, con la clase Persona se pueden formar los individuos JuanPérez, LolitaAyala y VicenteGuerrero, que son ejemplos de personas. También se puede concebir a una clase como el conjunto de todos los individuos que ha formado.

Herencia. Si ya definimos varios métodos y datos privados para un conjunto (una clase) de, digamos, personas, no vale la pena repetirlos para la clase clientes. Para que la clase Cliente herede de Persona todas sus ranuras y métodos, solo basta definir a Cliente (cuando se le crea) como un subconjunto de Persona. De la misma manera, los individuos heredan de la clase a que pertenecen las ranuras y métodos en ella definida; esto es lógico, puesto que Persona es un molde con métodos y ranuras; al formarse LolitaAyala, hereda todas éstas del molde.

Encapsulamiento, polimorfismo

Los datos (las ranuras) de un objeto no están accesibles a todos los programas (los métodos de otros objetos) en forma abierta o pública, sino a través de métodos o procedimientos para pedirlos. Por ejemplo, la forma de obtener el saldo del cliente número 18 no es mediante un acceso directo al campo donde el saldo está guardado, tal como JuanPérez.saldo. Es necesrio ejecutar un pequeño programa (llamado método) asociado con el cliente, para pedírselo: dame_saldo (JuanPérez). Con esto se gana que la forma en que la información esté guardada, solamente concierne al método dame_saldo, y los accesos al saldo se encuentran concentrados en un solo lugar (en dame_saldo). También es posible exigir otras verificaciones o cómputos antes de dar el saldo: tales cálculos adicionales simplemente van dentro de dame_saldo. Los métodos que invoquen a dame_saldo no necesitan saber de estas verificaciones adicionales. Inclusive, quizá, ¡tal vez la información no se encuentra guardada en el objeto (cliente) JuanPérez en ninguna forma, sino que es calculada cada vez que alguien la solicita!

versus "orientada a"

- **Programación con objetos**

  Las operaciones están asociadas a los objetos que las sufren

- **Abstracción**

  Los objetos se definen por un conjunto de propiedades (ranuras) y métodos (funciones) para cambiarlas, accesarlas, definirlas, etc.

- **Objeto**

  Es un conjunto de propiedades (con valores) y métodos para operarlas

  Es una estructura de datos y programas para manipularlas

- **Clase**

  Es un molde para fabricar objetos específicos (individuos)

  Se puede pensar que una clase es el conjunto de individuos que la forma. Una clase no es un objeto.

  - **Individuo**

    Es un objeto específico. Es un miembro de una clase

    Un individuo (un objeto) posee ranuras y métodos.

  - **Herencia**

    Las clases heredan sus propiedades (ranuras) de las clases superiores. Los individuos heredan sus ranuras de la clase a la que pertenecen

- **Encapsulamiento**

  Los programas que usan objetos no saben de éstos más que lo que los mismos objetos hacen público

- **Polimorfismo**

  Existen nombres genéricos de funciones, reemplazables por funciones específicas

2.1.3 Ranuras o propiedades

Una propiedad o ranura relaciona dos objetos, o un objeto con un dato primitivo (entero, cadena, número real; aquellos datos simples que vienen predefinidos en C++). Es decir, en cierta manera, los datos simples son objetos simplificados —se representan de una manera económica, eficiente para la máquina. Usando este punto de vista, en las ranuras solo hay objetos —aunque algunos de ellos tienen representaciones simples; pueden considerarse como detalles de implementación.

Una ranura tiene, además de su nombre, un dominio: la clase más general que aún puede poseer la ranura; para la cual aún tiene sentido esa ranura. tiene además un rango: la clase más general cuyos individuos pueden ser el valor de esa ranura. Si pensamos en una ranura como en una flecha o apuntador dirigida de un objeto (el dueño de la ranura) a otro (el valor de la ranura), entonces el rango es la clase del objeto que está al principio, y el dominio la que está al final. Tienen las ranuras también una cardinalidad: el número de elementos de su rango.

En C++, una ranura no es un objeto. En otros lenguajes, como CYCL (sección 2.4), una ranura es un objeto, que tiene a su vez varias ranuras importantes: `nombre`, `tiene_sentido_para` (es decir, el dominio), `mis_valores_son` (esto es, el rango), `cardinalidad`. Las ranuras forman jerarquías de especialización: `buen_padre` es una especialización de `padre`, que lo es de `pariente`, que lo es de `conoce_a`.

Desde el punto de vista lógico, una ranura es una afirmación (un hecho, constante, hipótesis) que se hace entre dos objetos. Cuando `AdolfoGuzmán` tiene `edad` con valor `60`, estamos afirmando que entre el objeto `AdolfoGuzmán` y el objeto `60` existe la relación `edad`. Por consiguiente, las ranuras con valores múltiples se consideran como la conjunción ("Y") lógica de afirmaciones sencillas: `AdolfoGuzmán amigo_de (JuanPérez JoséLópez)` significa que hay dos relaciones `amigo_de`, una de `AdolfoGuzmán` hacia `JuanPérez`; la otra de `AdolfoGuzmán` hacia `JoséLópez`. Ambas existen, ambas son ciertas.

La inversa de una ranura es otra ranura con dominio y rango intercambiados. Es una flecha en la dirección opuesta. Por consiguiente, la inversa de la inversa es la ranura original. es importante que todas las ranuras tengan inversa, ya que así es posible navegar de cualquier objeto a cualquier otro. De lo contrario, hay que hacer búsquedas en conjuntos. Ejemplo: Cada persona tiene una ranura `periódico_que_lee` que apunta a `Excélsior`, `El_Universal`, etc. Sin embargo, un periódico no tiene la ranura inversa `mis_lectores`. Entonces, podemos preguntarle a `AdolfoGuzmán` qué periódico lee, y nos contestará: `Excélsior`. Sin embargo, la pregunta ¿quiénes son los lectores de Excélsior? se tiene que contestar haciendo una búsqueda entre todas las personas, fijándonos quién tiene a `Excélsior` en su ranura `periódico_que_lee`. Esta búsqueda se hubiera evitado con la existencia de la ranura `mis_lectores` en cada periódico; esa ranura tendría la respuesta.

Las ranuras que tienen objetos simples como rango no pueden tener inversa, pues estos objetos simples (números, cadenas) no son en realidad objetos y no pueden tener ranuras.

- Ranuras o propiedades
  - Valor
  - Valor por omisión
  - Verificación automática del tipo (clase) de un valor
  - Representación en memoria
- Afirmación lógica
  - Una propiedad (ranura) puede verse como una afirmación lógica

    (edad Guzmán 60)

    (lugar_de_nacimiento Benito_Juárez Guelatao)
- Inversa ↕ dar ejemplo
- Propiedades de las ranuras
  - No son objetos. No en C++. Sí en CYCL.
  - Dominio

    La clase más general para la cual están definidas. Dominio de la ranura edad: Persona. O mejor, Animal.
  - Rango

    La clase más general de la cual pueden tener individuos como valor. Rango de la ranura edad: números no negativos.
  - Cardinalidad (ó aridad)

    Ranuras con un solo valor: longitud, peso. Con varios valores: amigo, hijo, dueño_de.
  - Soporte de un valor

2.1.4. Valores. Cómo y cuándo se computan

Los valores de una propiedad son otros objetos. Cómo se computa el valor de una propiedad o ranura es algo interesante.

Valor local. la mayoría de las propiedades tienen un valor porque el usuario lo puso ahí. Desde el punto de vista lógico, el usuario hizo la afirmación que la edad de AdolfoGuzmán era 40 cuando puso el 40 en la ranura edad. No todos los objetos a los que les está permitido tener edad tienen un valor para ella; podemos, por ejemplo, no conocer la edad de JuanPérez. De la misma manera que no a todas las personas tienen un elefante, aunque todas tienen el derecho u oportunidad de tenerlo.

Valor por omisión. El valor por omisión lo da la clase más cercana a la que el individuo pertenece. Por ejemplo, si ponemos en la clase `Mexicano bebida_alcohólica (Tequila)`, estamos diciendo que, por omisión, a falta de mayores conocimientos, si una persona es mexicana, bebe tequila. Ahora fijémonos en un mexicano en particular, digamos Don Lázaro Cárdenas. ¿Bebe él tequila? Sí, por omisión, cuando no sabemos más de él. Bueno, pero a mí me consta que don Lázaro bebía Charanda. ¿Hay problema o contradicción? No; simplemente agregamos `Charanda` y ya se tiene `LázaroCárdenas bebida_alcohólica (Tequila Charanda)`. ¿Y si sé que él no bebía tequila? Lo borramos y nos queda `LázaroCárdenas bebida_alcohólica (Charanda)`. Porque tenemos información local (particular) de que don Lázaro no bebía tequila, aunque suponemos que todos los mexicanos (por omisión, a falta de mejor información) lo beben.

Valor por inversa. Tan pronto ponemos un valor en una ranura, el valor de la ranura inversa está determinado, y el sistema debe ponerlo automáticamente (CYCL así lo hace; en C++ hay que invocar explícitamente una función para esto). Sea `me_beben` la ranura inversa de `bebida_alcohólica`. Tan pronto como dije que LázaroCárdenas bebe charanda, aparece él en la ranura `me_beben` de Charanda. Queda así: `Charanda me_beben (LázaroCárdenas JuanPérez JuanitaCruz ...)`.

Consecuencia. Un valor puede aparecer como consecuencia de otro. Estos valores los ponen los demonios o procesos asíncronos (o funciones llamadas dentro de los métodos de la ranura que influye en las otras, la que provoca la consecuencia). Si digo (se lo expreso al sistema a través de un demonio) que tener agruras es una consecuencia de haber comido demasiado, al agregar a `AdolfoGuzmán cantidad_de_alimentos_ingeridos (Mucho)`, el sistema (el demonio que yo crié) le agrega a `AdolfoGuzmán percepción_física (Agruras)`.

Cuándo se calcula el valor

Evaluación hacia adelante. Cuando se tiene. Cuando se guarda, cuando se deposita. Es lo más común. En cuanto cambia de valor la variable de la cual depende éste. Si el diámetro es dos veces el radio, en cuanto el radio adquiere el valor 3, el diámetro se calcula y adquiere el valor 6.

Evaluación perezosa. Cuando se pide, cuando se quiere leer el valor del diámetro. Entonces se toma nota de que el radio vale 3, y se hace el cálculo. Tiene la ventaja de no hacer muchos cálculos innecesarios, si el valor del diámetro es solicitado esporádicamente.

- Valores
  - Cómo se computa el valor de una propiedad
    - Valor local
    - Valor por omisión
    - Valor por inversa
    - Consecuencia, deducción
    - Valor típico. Se toma de algún "hermano" cercano
    - Valor por tarea o agenda
    - En general, valores computados por demonios, reglas o procesos asíncronos
  - Cuándo se computa el valor de una propiedad
    - Cuando se guarda o deposita. Lo más común; valor local
    - Cuando se cambian los valores de otras ranuras de las cuales depende

      Propagación hacia adelante, o provocada por los datos ("data driven").
    - Cuando se necesita su valor

      Propagación hacia atrás, evaluación perezosa, o provocada por la demanda ("demand driven").
    - Diferencias entre las anteriores. Podemos mezclarlas
  - Qué sucede con el valor de una propiedad
    - Se guarda una vez conocido

      "Caching". Ventaja: rápido de accesar después
    - Se descarta: se calcula cada vez que se necesita

      Siempre se tiene el último valor. Se usa para compartir valores entre procesos concurrentes, para usar valores "dinámicos" que están cambiando: número de asientos disponibles en el vuelo de avión MX 437.

2.1.5 Métodos

Cada metodo es el nombre de una función u operación asociada con el objeto. Estas funciones operan o modifican a los valores de las ranuras del objeto asociado, aunque también pueden afectar a otros objetos. Los métodos llevan argumentos que indican precisamente qué o cuánto hay que modificar o hacer con el método en cuestión y el objeto en cuestión. Por ejemplo, su el objeto Cliente_13 tiene un método agrega_a_mi_saldo, entonces el mensaje agrega_a_mi_saldo 300, enviado al objeto Cliente_13, le agregará 300 pesos al saldo del objeto en cuestión.

Un método es un programa escrito en el lenguaje de objetos, que es un lenguaje textual. Si este lenguaje permite escapes a otros, es posible tener un método definido en otrolenguaje. Porf ejemplo, en C++, los métodos generalmente están escritos en C++ (o en C, que es un subconjunto de C++), pero puede haber uno en Ada, ya que C permite un escape a Ada.

Asimismo, un método puede definirse en SQL, para accesar una base de datos relacional. Es posible, pues, definir que el valor de cierta ranura se compute mediante un método que, en efecto, lee ese valor de una base de datos. De esa manera se obtiene un acceso transparente a la base de datos, pues el usuario solo invoca el método del (solo "manda un mensaje" al) objeto, y pronto se olvida de que se están leyendo valores de la base de datos.

Los métodos también pueden cambiar la forma de despliegue del objeto, si éste tiene alguna representación visual. De esta manera, puede crecer de tamaño, cambiar de color, etc.

En esta forma se obtienen interfaces a "cosas externas" a los objetos, como una pantalla de despliegue, un termómetro que mide la temperatura del aire.

Es posible que los métodos tengan un número variable de argumentos. También es posible especificar valores por omisión para algunos (o todos) los argumentos.

Polimorfismo

Varias clases pueden tener métodos definidos con el mismo nombre. El compilador decide, fijándose en el tipo de argumentos, cuál de los diferentes métodos escoger.

- **Métodos**
  - **Programas en C++**

    Programas en el lenguaje donde los objetos se definen
  - **Escape a otros paradigmas de programación**
    - **Programas en otro lenguaje de programación**

      Nos permiten utilizar programas viejos
    - **Ranuras que se definen accesando una base de datos**

      Dan a la ranura valores guardados en la base
    - **Llamadas a SQL**

      Advertencia: SQL regresa un conjunto de valores, no uno solo
    - **Llamadas a interfaces de entrada y salida**

      Para desplegar

      Para medir la temperatura del aire
  - **Los métodos disponibles pueden depender del usuario**

    Ejemplo: El medio ambiente BackPlane de Atherton. El acceso a determinado métodos depende del tipo de usuario. Si es programador, documentador, probador de versiones, etc.
  - **Los métodos pueden tener argumentos opcionales**
    - **... y número variable de argumentos**

2.1.6 Herencia

La herencia es la asignación automática de ciertas cosas en virtud de que el heredero cumple con cierta propiedad. En este caso, un objeto hereda los métodos, las ranuras y sus posibles valores por omisión de la clase a la cual pertenece.

Las clases heredan los métodos y ranuras de sus super-clases. Los individuos heredan sus ranuras de las clases a las cuales pertenecen. Es decir, se hereda a través de la relación subclase - subclase - ... - miembro. Por ejemplo, el hecho de haber definido la ranura `número_de_patas` con rango en la clase Animal, faculta a cualquier miembro de esa clase a poseer tal ranura. También si se da en la clase Animal un valor por omisión a esta ranura (4 patas, digamos), entonces todo animal "nuevo" tendrá 4 patas. Este valor por omisión puede descartarse fácilmente con solo asignar un nuevo número: 2 patas, para las aves. En este sentido, el valor por omisión es un valor "débil", fácilmente desechable por otros valores. Su interpretación es la siguiente: si no tenemos más evidencia acerca del número de patas de un animal particular, entonces asumamos que tiene cuatro (el valor por omisión). Por ejemplo, por omisión, todos los elefantes tienen cuatro patas. Sin embargo, si sabemos que Clyde tiene 3 patas, el 3 "gana" al 4.

Herencia sencilla

Cuando un individuo pertenece a una sola clase inmediata, solo hereda de esa clase.

Herencia múltiple

Cuando un individuo (un objeto) pertenece a más de una clase inmediata, hereda ranuras (y métodos) de ambas.

Si un individuo tiene herencia múltiple y hereda distintos valores por omisión sobre la misma ranura, ¿qué sucede? Por ejemplo, JuanJiménez es Marino, y sabemos que todos los marinos (por omisión) beben bebidas alcohólicas. También sabemos que JuanJiménez es Sacerdote, y sabemos que los sacerdotes (por omisión) no beben alcohol. ¿Qué hereda JuanJiménez por omisión, puesto que los valores son contradictorios? En cierto sentido, estamos en un empate. El compilador simplemente reporta que no puede decidir cuál de los dos valores heredar a JuanJiménez, y le pide al usuario que él decida.

El problema no es de fondo. En primer lugar, tal vez sepamos que, de hecho, JuanJiménez no toma. Al introducir este hecho local, el valor por omisión (sea el que fuere) queda descartado, sin que haya contradicción alguna.

El mismo conflicto sucede si se heredan métodos distintos pero con el mismo nombre de las diferentes clases a las cuales el objeto pertenece. El compilador marca una ambigüedad y le pide al usuario que la resuelva.

- Herencia
  - Una clase hereda sus ranuras de su superclase inmediata superior
    - Y así recursivamente
    - Por consiguiente, hereda sus ranuras de todas sus superclases
    - También hereda los valores por omisión
  - Un objeto (individuo) hereda sus ranuras de su superclase inmediata superior
    - También hereda los valores por omisión
- Herencia sencilla
  - Cuando un objeto pertenece a una sola superclase inmediata superior
- Herencia múltiple
  - Cuando un objeto pertenece a más de una superclase inmediata superior
  - A veces hay conflictos con la herencia de valores por omisión
- Herencia a través de cualquier ranura
  - No existe en C++
    - Ejemplo: el apellido se hereda a través de la ranura hijo_de. Un mal olor se hereda a través de la ranura vecino_de.
- Cuándo se hereda — en el momento de crear al individuo

2.1.6. Propiedades derivadas o computadas

Que una propiedad tenga un valor (o varios) guardados en ella es solo una manera de darle valor a una propiedad o ranura. Es la forma más común de darle valor a una ranura.

Otra manera consiste en definir una ranura como computada, es decir, que su valor depende de cierto cómputo o fórmula asociado a la ranura, en vez de tener un valor pre-almacenado en ella.

Conviene definir como computadas aquéllas ranuras que son auxiliares, o que dependen de otras y que es conveniente tomar como variables dependientes, reservando para otras ser ranuras "independientes". De la fórmula diámetro = 2 x radio, ¿cuál es la variable independiente, el radio o el diámetro? Cualquiera puede serlo. Es cuestión de preferencias.

También conviene definir como propiedades computadas aquéllas que se usan infrecuentemente, ya que si las definiéramos como de "valor guardado", sería fácil olvidarnos de actualizarles su valor. Es mejor que se actualicen automáticamente, por ser propiedades derivadas.

Otra ocasión donde conviene definir una ranura como computada es cuando su valor cambia constantemente pero es raramente utilizado. En este caso, va a ser una ranura computada "de propagación hacia atrás" o computada por demanda (sección 2.1.4). Por ejemplo, el total de todos los saldos de todos mis clientes. Voy a dar una fórmula para calcularlo, en vez de estar actualizando la ranura cada vez que un cliente viene a hacer un movimiento a su cuenta. Pero voy a hacer algo más: la defino como "computada bajo demanda", para que no se calcule cada vez que un cliente entra o sale sino que se compute solamente cuando yo la pido.

Las ranuras computadas "bajo demanda" tienen propiedades agradables: no ocupan lugar en la memoria (si descontamos el pequeño lugar que la fórmula ocupa), y ¡están siempre actualizadas! Pero tienen un defecto: son lentas para darnos su valor. Porque lo tienen que computar a la hora pedida. Si la necesito en este momento, tengo que esperar a que la suma de todos los saldos individuales se efectúe. Esto puede no ser una espera demasiado molesta, pero si estamos escribiendo programas en tiempo real, la espera puede ser excesiva. Es conveniente definir una ranura como computada bajo demanda cuando su consulta es mucho menos frecuente que su actualización.

Propagación de valores

Cuando hay ranuras computadas, éstas pueden formar cadenas y provocar que varias se actualicen a la vez. Empero, la actividad es transitoria (como las ondas en un lago) y las ranuras vuelven pronto a estar estáticas (sin cambio de valores). La razón es sencilla: el número de ranuras que pueden cambiar como reacción al cambio en una de ellas es finito, y, una vez que todos los cambios se efectúan, no hay más que hacer.

Una excepción a lo anterior es cuando hay inconsistencias, lo que hace que el proceso no converja. Ejemplo: reglas o fórmulas contradictorias. O converja muy lentamente: ejemplo: el diámetro es dos veces el radio. El radio es la mitad del diámetro. Pero, si por errores de redondeo, el doble de 3 no es 6, sino 5,99999, y la mitad de 5,999999 sufre también de un redondeo, los valores pueden ir disminuyendo lentamente hacia cero.

- **Propiedades derivadas o computadas**
  - **Cuándo conviene definirlas**
  - **Cómo están implementadas**
    - **Con demonios o procesos asíncronos**
    - **Los demonios están definidos, en vez de asíncronamente, junto o dentro de las ranuras de las cuales dependen**

      Las ranuras independientes
    - **Hay ranuras que se definen atadas a la realidad**

      Su valor depende de la temperatura de una caldera, por ejemplo
    - **Otras se definen accesando una base de datos**

      Su valor depende de las ventas de zapatos registradas en Oracle
  - **Recordando los valores de las ranuras**
    - **Las ranuras conservan su valor en forma local**

      En memoria principal
    - **Hay ranuras que se reflejan a disco**

      * Write-through, o escritura profunda

      * Se usan para actualizar valores dinámicos: el número de asientos disponibles en el vuelo Mexicana 347.
- **Propagación de valores**
- **El problema de los errores de redondeo**
- **El problema de ciclos inherentes**
  - **Si tienes una silla verde, píntala de rojo**
  - **Si tienes una silla roja, píntala de verde**

2.2 Demonios

Un demonio es un proceso (un programa) asíncrono, que es disparado (se ejecuta) cuando los datos (valores en ranuras. ¿Cuáles ranuras? El demonio las define; son sus "argumentos") a los que está asociado cambian de valor. Por extensión, existen demonios que son disparados a cierta hora, o con cierta frecuencia. Se pued pensar que ellos están ligados a la variable "tiempo", o al reloj.

Un demonio posee dos partes: su parte izquierda, predicado, guardia o función de disparo, que, cuando se vuelve cierta, hace que su parte derecha se ejecute. [Si su predicado no es cierto, el demonio termina su ejecución, y vuelve a ser disparado o ejecutado cuando vuelve a cambiar un valor en alguna ranura a la que él está asociado. Entonces tendrá otra oportunidad de evaluar su parte izquierda o predicado]. La parte derecha del demonio es una acción, por ejemplo, cierta otra ranura adquiere un nuevo valor.

¿Qué tan a menudo se ejecuta esa parte izquierda? Desde un punto de vista lógico, "tan a menudo como fuere posible". En realidad, la implementación no se efectúa haciendo trabajar al demonio muchas veces (auscultando él los datos), sino que se hace colocando una interrupción en la ranura y clase correspondiente. O más sencillo aún: el código que va a cambiar la ranura llama al demonio asociado antes de hacer efectivo el cambio. Esto tiene el efecto de que solo cuando se trata de cambiar lo que va en esa ranura es necesario consultar al demonio ahí colocado.

2.2.1 Restricciones

Una restricción es una limitación a ciertas propiedades. Cierta combinación o fórmula entre los valores de algunas propiedades debe cumplirse siempre. Por ejemplo, si un objeto tiene valor para su ranura diámetro y también para su ranura radio, estos valores deben estar en la relación $d = 2\,r$. La forma en que se define una restricción es un demonio cuyo predicado es la negación de la fórmula que siempre se debe cumplir. En tanto que la fórmula se cumpla, el predicado es falso y la acción del demonio-restricción no se dispara. Cuando la fórmula no se cumple, el demonio dispara su acción. Ésta consiste en dar un aviso o mensaje de error, y pedir un nuevo valor para el valor de la última ranura, que se considera que fué la que "precipitó" que la fórmula a vigilar no se cumpliera. Si una ranura tiene más de un valor, la restricción se aplicó a cada uno de ellos (conforme fue añadido a la ranura).

Cualquier intento de agregar o modificar algún valor a alguna ranura sobre la cual una restricción está definida, es verificada por esta restricción, "para ver si le da permiso de proceder."

Es conveniente atar las restricciones en el lugar más específico posible que aútenga sentido para esa restricción. De esta manera se logra una ejecucióeficiente de los demonios, ya que solo las propiedades de las clases específicas (las más pequeñas posibles) son las que consultarán al demonio (pidiéndole que evalúe su parte izquierda o predicado) cuando cambien. Cambios en las demás ranuras, o en las mismas ranuras pero de una clase más genérica —para la cual la restricción ya no impera— no provocarán pérdida de tiempo consultando a un demonio que es claramente irrelevante. Por ejemplo, si decimos que un miembro de la clase PersonaMayordeEdad debe tener más de 65 años, conviene tener tal restricción sobre PersonaMayordeEdad, y no sobre Persona o Animal.

Partición. Una partición de un conjunto son varios subconjuntos del mismo, de tal manera que son mutuamente excluyentes y colectivamente exhaustivos. Se implementa su verificación usando restricciones. Ejemplo: Persona tiene como partición {Hombre, Mujer}.

- **Demonio**
  - Proceso que ejecuta su lado derecho (acción) cuando su lado izquierdo (predicado) se satisface
- **Restricción**
  - Demonio cuya acción es quejarse y no permitir que cierto valor se deposite, cuando su predicado se satisface
    - "Un comprador de bebidas alcohólicas debe tener 21 o más años"

      Esta restricción, aplicada a la ranura comprador de la clase BebidaAlcohólica, verifica la edad del posible comprador y se queja si tiene menos de 21 años.
    - El valor "malo" no entra a la ranura. No se guarda
    - Caso especial: cuando un valor no está definido

      Una respuesta de "no sé", o sea la ausencia de información para evaluar el predicado, no provocará que éste se satisfaga.

      Es decir, si no sé la edad de JuanPérez, no debo quejarme "tiene menos de 21 años." No me consta.
  - Colocación de las restricciones
    - En el lugar más específico posible

      Ejemplo: El diámetro debe ser dos veces el radio
  - Cuidado con las condiciones "si y solo si"
    - Ejemplo: "x pertenece a la clase Hombre si y solo si es Persona y su sexo es masculino"

      Primero creo a JuanPérez como Persona. Luego le doy sexo masculino. En este momento, la restricción se quejará de que cómo es posible que Juan tenga sexo masculino, si no pertenece a la clase Hombre.

2.2.2. Métodos de propagación

Propagación hacia adelante, o disparada por los datos

En cuanto se conocen todos los valores de "entrada" (valores independientes) de un demonio, el valor que éste calcula en su parte derecha en efecto se calcula y se almacena en la ranura "dependiente". Esto hace que el valor esté actualizado y listo o disponible, por lo que su acceso es rápido.

Si el nuevo valor es usado como entrada (como valor independiente) de algún otro demonio, se dispara también este otro, de manera que un cambio en una variable puede ocasionar una cadena de cómputos. Esta cadena eventualmente terminará (salvo que sea inconsistente), puesto que solo hay un número finito de ranuras que actualizar.

Propagación hacia atrás, demorada, perezosa, computada bajo demanda

En este caso, en cuanto se conocen todos los valores de "entrada" (valores independientes) de un demonio, se marca como inválido el valor actual (si existe) de su ranura independiente. Es decir, hay nuevos datos que nos hacen sosperchar que este valor ya no es vigente, pero no queremos computarlo en este momento.

Cuando se accesa para lectura a este valor se ve que no es inválido. Entonces, se ve si hay demonios asociados. Sí hay uno, el perezoso. Se le echa a andar. Y nos da el valor que queremos. Y se marca como válido o vigente.

Ejemplo: GT, el total de todos los saldos de todos mis clientes. Voy a dar una fórmula para calcularlo. Defino esta fórmula como computada bajo demanda, para que no se compute cada vez que un cliente hace un movimiento a su cuenta bancaria. Se computará solamente cuando yo deseo conocer(leer) GT.

Propagación hacia el disco

Cuando cambio un valor a una ranura en memoria, ningún otro proceso se entera. La razón es clara: otro proceso no puede leer mi memoria. Esto es a menudo lo que se desea: yo puedo estar cambiando el sueldo a un empleado, o haciendo otro tipo de experimentos o simulaciones, sin afectar a otros procesos, sin que otros procesos se enteren o percaten de mis cambios.

En ocasiones, es importante informar a los otros procesos de los cambios que yo hice en cierta variable. Ejemplo: en el Vuelo347 de la aerolínea Mexicana de Aviación, la ranura asientos_disponibles. Si tiene un valor de 23 y vendo yo dos asientos, además de actualizar el valor 21 en asientos_disponibles en mi memoria, debo ir al disco y actualizar el valor 21 ahí. Con el objeto de que otros procesos tomen en cuenta el número de asientos que vendí. Es responsabilidad del manejador de objetos en disco ir a informar a los otros procesos de este cambio. En primer lugar, el manejador debe saber cuáles otros procesos copiaron el objeto Vuelo347 a memoria principal. El aviso puede darse de dos formas: como propagación hacia adelante, actualizando en cada memoria el nuevo valor 21; o como propagación hacia atrás, indicando en cada ranura simplemente que el valor de ella ha quedado inválido (obsoleto).

- **Métodos de propagación**
  - **Propagación de ranuras**
  - **Propagación de valores por omisión**
  - **Propagación de valores computdos**
    - **Propagación hacia adelante (provocada por los datos)**
    - **Propagación hacia atrás (provocada por la demanda)**
  - **Propagación hacia el disco**
    - **Nos permite que otros procesos (otros programas activos) usen el valor que acabamos de asignar a una ranura**

      * Es una escritura profunda ("write through") a la ranura que se encuentra en disco

      * Se usa cuando tenemos valores que "residen" en disco y queremos que un nuevo valor computado sean compartidos por otros procesos
    - **Propagación hacia adelante**

      Una vez que la ranura en disco recibe el nuevo valor, la base de objetos va e informa del nuevo valor a todos los procesos que tienen copia de la ranura
    - **Propagación perezosa**

      Una vez que la ranura en disco recibe el nuevo valor, la base de objetos va e informa a todos lor procesos que tienen copia de la ranura, que el valor que ahora poseen en esa ranura es inválido o viejo. No les actualiza inmediatamente el nuevo valor. Cuando este valor se necesita, el objeto se percata de que su ranura tiene un valor inválido y va al disco a leer el nuevo valor

      Ventaja: estos valores siempre están actualizados

2.3 Principales postulados del lenguaje C++

El lenguaje de programación C++ es una extensión al lenguaje C, hecha para proveer objetos. Todas las construcciones de C están disponibles en C++. De hecho, el compilador de C++, en sus inicios, fue simplemente un preprocesador (similar a un expansor de macros) que tomaba un programa en C++ y lo convertía en otro equivalente en C.

Visibilidad

En C++ se puede controlar lo que deseamos que otros objetos van (la parte pública: métodos y ranuras) de nosotros. Hay un control de acceso. Si no se especifica si una ranura es pública o privada, por omisión es privada.

Herencia

En C++, la herencia se lleva a cabo a través de la relación subclase - subclase - ... - individuo, y nadamás a través de ella. Una clase hereda a todas sus subclases todas sus ranuras y métodos. Y finalmente, una clase hereda lo mismo a los individuos que se van formando. El individuo hereda de su clase inmediata (la más próxima a él, la que lo creó), pero ésta ya contiene todas las ranuras y métodos procedentes de clases superiores. En los compiladores iniciales de C++, la herencia es sencilla. Actualmente, existe herencia múltiple. Ahora puede haber varias cadenas subclase - subclase - ... - individuo, para un individuo en particular.

Adelantos de C++ sobre C

Las funciones en C++ pueden tener un número variable de argumentos. A éstos se les puede pasar un valor por omisión, si no están presentes en la se4cuencia de lllamado.

Es posible tener en C++ funciones polimórficas (sobrecargadas).

• Principales postulados del lenguaje C++

■ Clases en C++

- Datos miembros (ranuras)
- Funciones miembros (métodos)
- Visibilidad: Público Privado Amigo
- Llamadas implícitas: Constructores Destructores
- Miembros: Estáticos Const

■ Herencia en C++

- Clases derivadas
- Funciones virtuales
- Llamadas implícitas
  - * Constructores de base * Destructores virtuales
  - * Orden en las llamadas
- Visibilidad
  - * Derivación pública * miembros protegidos

■ Mejoras de C++ sobre C

- Funciones
  - * argumentos opcionales * valores por omisión
  - * Sobrecarga de funciones * prototipos y verificación del tipo
  - * Funciones en línea
- Constantes
  - * Escalares * Apuntadores * Apuntador a una constante
- Tipos de referencia
  - * Variables * Parámetros * Inicialización * Uso
- Conveniencias sintácticas
  - * Comentarios * Declaraciones dentro de bloques

2.3.1 Conceptos

Con cierto detalle se verán los principales postulados del lenguaje C++. Este es un lenguaje implementado como una extensión a C. Los compiladores iniciales traducían un programa escrito en C++ a otro escrito en código C, el que despues se puede llevar (por el compilador C) a código objeto. Ahora existen compiladores como el de Zortech Inc. para Ms-Dos que producen directamente código objeto, sin traducirlo primero a C.

Aunque el lenguaje tiene algunos postulados que pueden verse como un intento de arreglar lo que está mal en C, no es ésta la finalidad principal de C++; su verdadero poderío es en la introducción de clases de objetos, cuyos miembros se comportan de una manera similar. El poder de la programación con objetos causa hábito, y uno se acostumbra al uso de la encapsulación, herencia y polimorfismo. Sin embargo, C++ no es una panacea, y hay muchas aplicaciones que no se benefician con el uso de C++. No es automático, pues, que el C++ (o algun otro lenguaje orientado a objetos) haga obsoletos instantáneamente a todos los lenguajes de tercera generación.

¿Cuáles son las principales aplicaciones o tipo de programas que se benefician con un lenguaje de objetos? Curiosamente, esta pregunta no tiene ahora una respuesta clara, no todavía. Los lenguajes de objetos son relativamente jóvenes y no han encontrado sus nichos. Relativamente pocos conocen y utilizan C++, aunque este número va en aumento, sobre todo porque C++ es una novedad relativa y un lenguaje interesante, de moda. Segundo, no se han hecho muchos programas, ni muy extensos, usando C++. No estoy diciendo que los lenguajes con objetos no valgan la pena. Es más apropiado decir que "no ha comprobado su eficacia." Estamos en una época similar a la de la introduccion de APL, un lenguaje que "prometía mucho" pero que siempre no se popularizó. ¿Vale la pena aprender C++? Sí, un poco para estar a la moda, y un poco porque los conceptos que C++ maneja parecen lo suficientemente interesantes y poderosos como para prometer una larga vida a C++. Sin embargo, en caso de que C++ perdure, sin duda habrá cambios significativos a sus postulados y semántica actuales.

¿Qué nichos es probable que un lenguaje con objetos ocupe? Uno de ellos puede ser el diseño de interfaces gráficas, donde hay un claro beneficio en que un tipo de objeto (una pantalla para desplegar texto, por ejemplo) herede muchas características de otra clase más general (pantallas). Esto reduce la produccion de código, pues solo hay que escribir el código específico para la pantalla de datos, y heredar de la super-clase pantalla aquellas otras funciones (métodos se llaman en C++) que son comunes a todas las pantallas: abrirlas, cerrarlas, moverlas, cambiarles tamaño, etc. Probablemente C++ no desplazará al Fortran en aplicaciones de Ingeniería, sobre todo aquellas de mucho manejo de números y pocas estructuras. Probablemente para accesar bases de datos en forma un tanto independiente del manejador se siga usando SQL.

- Conceptos
  - Extensiones a C
    - Cast
    - Void
    - Tipos
    - El operador ::
    - new y delete
    - Funciones en C++
  - Clases
  - Referencias
  - Funciones
  - Sobrecarga y polimorfismo
    - Sobrecarga de funciones

      advertencia
    - Sobrecarga de operadores
  - Operadores
    - Operadores unarios
    - Operadores binarios
    - Operadores de asignación
    - Operadores de acceso a miembros
    - Operador para sub índices
    - Operador para llamar a funciones

2.3.2 Clases

Definición de clase

Una estructura (*struct*) en C contiene uno o mas campos (llamados *miembros*), que se agrupan como una sola unidad. En C++ la sintaxis de *struct* se extiende para permitir la inclusión de funciones (código) en las estructuras. Estas funciones definidas dentro de una estructura tienen una relación especial a los campos (o miembros) de esa estructura, y se denominan *métodos*. Nosotros preferimos llamar *ranuras* o *propiedades* a estos campos, y reservar el nombre *miembro* para decir que un objeto es miembro (o pertenece) a una clase, lo que se ve a continuación.

Objetos

En C++, un objeto es un ítem que se declara ser miembro de una clase, y se dice que es un objeto de tal tipo de clase (o de tal tipo). Por ejemplo, con respecto a la clase Persona, x1, un miembro de esa clase, es un *objeto*; específicamente, x1 es un objeto del tipo Persona. [En general, en un lenguaje orientado a objetos, los objetos pueden ser tanto elementos de una clase (llamados individuos) como clases mismas; empero, en C++ las clases no son objetos, sino son *tipos*.]

Un objeto tambien se denomina una instancia de una clase. Un apuntador o referencia a un objeto nos proporciona una manera indirecta de accesar tal objeto, pero el apuntador o referencia misma no es una instancia de una clase.

Miembros (ranuras, propiedades)

Los miembros (ranuras, propiedades) de un objeto pueden ser *enums*, campos de bits, enteros u otros tipos intrínsecos o definidos por el usuario. Tambien pueden ser otros objetos. Es decir, el valor de una ranura o campo puede ser un tipo intrínseco, o definido por el usuario, u otro objeto, siempre y cuando ya esté definido (es decir, que pertenezca a una clase ya definida, que sea de un tipo ya definido).

Una clase no puede pertenecer (ser miembro) de ella misma. Sin embargo, una clase puede contener referencias y apuntadores a instancias de ella misma. Por ejemplo, la clase Oaxaqueño puede contener en la ranura distinguidos_hijos a las instancias BenitoJuárez, PorfirioDíaz, JoséVasconcelos; cada uno de estos objetos es una instancia de la clase Oaxaqueño.

- **Clases**
  - **Definiciones**
  - **Objetos**
  - **Miembros**
  - **Variables de la instancia**
    - **Uso**
    - **Cada variable de instancia es distinta**
    - **Se refieren a la instancia, aunque se definen en la clase**
    - **Son auto-contenidas**

      Se puede cambiar el valor de una sin afectar el valor de otras variables de esta instancia o de otras instancias (otros objetos)
  - **Métodos**
    - **Cómo declarar un método**
    - **Los métodos permiten acceso controlado a las ranuras del objeto**
  - **Constructores**
  - **Destructores**
  - **Objetos dinámicos**
  - **Amigos**

El objeto implícito

Una llamada a un método se asocia con objeto específico de dos maneras. La mas común es que la llamada al método contenga el nombre del objeto "dueño" del método, separado por :: Ejemplo: AdolfoGuzmán::draw("yellow") llama al método "draw" del objeto AdolfoGuzmán. La otra manera es no dar el nombre del objeto. Ejemplo: draw ("yellow") se refiere al objeto implícito. Es decir, a este objeto (el que está ejecutando o llamando al método). A este objeto se le da el nombre de *this*. this es una palabra reservada. Si la instrucción draw("yellow") se encuentra dentro del objeto AdolfoGuzmán, entonces se podría haber escrito también AdolfoGuzmán->draw("yellow") o this->draw("yellow").

Extensión de una clase

El termino extensión o alcance se refiere al area en un programa donde cierto identificador es accesible. Los dos alcances más comunes en C son *local* y *global*. Los identificadores declarados afuera del cuerpo de una funcíon tienen alcance global, queriendo decir con eso que pueden ser accesados de cualquier lugar dentro de ese programa. Si un identificador se define dentro de una función o de un bloque { }, será accesible solamente dentro de tal función o bloque, y se dice que su alcance es *local*.

El propósito del alcance es controlar el acceso a identificadores. C++ introduce el concepto de alcance de clase. Todos los miembros de una clase están dentro del alcance de esa clase; cualquier miembro de una clase puede hacer referencia a cualquier otro miembro de la misma clase. Esto es parte del **encapsulamiento**, puesto que C++ considera a todos los miembros de una clase como partes relacionadas de un todo.

Las funciones miembro (métodos) de una clase c tienen acceso irrestricto a las ranuras de tal clase. El acceso a ranuras y métodos de una clase fuera del alcance de la clase c está controlado por el programador. La idea es encapsular las estructuras de datos y la funcionalidad de una clase, de tal suerte que el acceso a ellas sea limitado o innecesario.

Especificadores de acceso

En una definición de clase, se usa un especificador de acceso (*public* o *private*) para controlar la visibilidad de las ranuras de una clase afuera del alcance de tal clase. Cualquier ranura o método listado en una sección *pública* de una clase puede ser accesada desde cualquier parte del programa. Aquellos listados en una sección privada (*private*) solo serán accesibles dentro del alcance de esa clase. Una clase puede tener varias secciones públicas y privadas. Tampoco se requiere un especificador de acceso; si se omite, todas las ranuras (y métodos) son privados.

- Objeto implícito

  draw ("yellow");

  AdolfoGuzmán->draw ("yellow");

  this->draw ("yellow");

- Alcance de una clase
  - Alcance en C:

    local

    global
  - propósito: controlar el acceso a ranuras y métodos
  - Encapsulamiento
  - Alcance de una clase:
    - Todos los miembros (elementos) de una clase están dentro de su alcance, y pueden referirse libremente entre sí.
    - Los métodos de una clase tienen acceso irrestricto a las ranuras de la misma clase.
- Especificadores de acceso
  - *public*
  - *private*
  - Una clase puede contener ninguna, una, o varias secciones públicas y privadas

    Por omisión, si no se declara pública o privada, se consideran privadas las ranuras y métodos

Constructores

Un constructor es un método especial que construye objetos. Llamamos a un constructor cuando queremos asignar espacio a un objeto, asignar valores a sus ranuras, y realizar otras labores administrativas para un nuevo objeto. Casi cualquier clase que se cree tendra uno o mas constructores. Para escoger cual constructor llamar, el compilador compara los argumentos usados en la declaración de un objeto con las listas de parámetros de los constructores. Es el mismo proceso que se usa para escoger entre funciones sobrecargadas. Si no se define constructor alguno para una clase, el compilador define un constructor por omisión, que carece de argumentos y simplemente llena de ceros cada byte de cada ranura. Esto es de poco uso, ya que un objeto asi inicializado generalmente no está preparado para funcionar como tal.

El constructor copy

Es un constructor que crea un nuevo objeto a partir de uno ya existente. Tiene un argumento: una referencia a un objeto de la misma clase. Existe un constructor *copy* por omisión (si no se declara un constructor *copy* específico), que copia el objeto donador bit por bit en el objeto destino. Si el objeto no tiene apuntadores o referencias a otros objetos en sus ranuras, tal copia es a menudo adecuada.

Destructores

Un destructor es un método con el mismo nombre que el de la clase, mas una tilde ~ al principio. Una clase tiene solo un método destructor, que no tiene argumentos y no regresa ningun valor.

Objetos dinámicos

Son objetos que se localizan en la pila, exactamente como otras estructuras de C. Los objetos dinámicos tienen la misma sintaxis de acceso a las ranuras que las estructuras dinámicas.

Una clase puede redefinir sus propias versiones de los operadores *new* y *delete*, con el fin de crear (y destruir) objetos en ciertos lugares de memoria, y soportar esquemas de asignación de memoria que varíen segun la clase. Esto debe manejarse con cuidado, preferiblemente por programadores ya experimentados en C++.

Amigos

Una clase puede otorgar a otro método o clase acceso privilegiado a sus areas privadas. Tal acceso debe ser explícitamente otorgado declarando a la otra clase o método como un amigo, mediante la declaración *friend*. Estos amigos son tratados como si fueran miembros de la clase, y poseen acceso irrestricto a las areas públicas de los objetos.

- Constructor
  - Método especial que construye objetos
  - Para una clase, en caso de ambigüedad, el compilador usa una manera específica para escoger constructor
    - Comparando los tipos de argumentos y cuantos hay
    - Es similar a como se resuelve la sobrecarga en llamados a funciones
  - Se usa un constructor por omisión si ninguno se define — de poca utilidad
- constructor copy
  - Crea un nuevo objeto a partir de uno ya existente
  - Existe uno por omisión — de bastante utilidad para objetos cuyas ranuras no contienen (apuntan) a otros objetos
- destructores
- objetos dinámicos
  - se localizan en la pila
  - su uso requiere experiencia
- amigos
  - para permitir que una clase o método tenga acceso al area privada de otra clase

2.3.3 Herencia y polimorfismo

Estos dos conceptos proporcionan el poderío completo de la programación orientada a objetos. La herencia permite poder construir nuevas clases sobre clases ya existentes. El polimorfismo trata objetos miembros de clases similares de una manera genérica.

Herencia sencilla; clase base

Una clase que hereda de otra se llama una *clase derivada*. La clase de la cual se hereda es la *clase base*.

Cualquier clase puede ser una clase base. Mas aun, una clase puede ser clase base para varias clases derivadas. A su vez, una clase derivada puede ser clase base para otra clase.

Los constructores y destructores no se heredan hacia las clases derivadas. En vez de ésto, los constructores para clases derivadas deben tener información (en forma de parámetros) para los constructores de la clase base. Dicho de otra forma, las llamadas a un constructor de clase base se hacen en la misma forma como se hacen las llamadas a los constructores para objetos miembros. Es decir, el nombre de la clase base se da, seguido por la lista de parámetros para el constructor.

Referencia a una ranura en la clase base

Una clase derivada no puede accesar los miembros privados de una clase base. Esto es un mecanismo de control puesto a propósito para prevenir que una nueva clase burle el estado privado de ciertas ranuras de la clase base.

Todas las ranuras en la parte pública de la clase base están accessibles dentro del alcance de una clase derivada.

Si una clase derivada declara que una clase base es pública, todas las ranuras (y métodos) públicas de la clase base se convierten en ranuras públicas de la clase derivada. Si una clase derivada declara que una clase base es privada, todas las ranuras y métodos públicos de la clase privada se convierten en métodos privados de la classe derivada. Desde luego, esto solo afecta al acceso a las ranuras públicas de la clase base desde afuera del alcance de la clase derivada.

Conversiones de clase

Un objeto (un miembro) de una clase derivada puede automáticamente usarse como si fuera un objeto de su clase base. Esto es debido a que la clase derivada contiene todas las ranuras de la clase base. Sin embargo, un objeto de una clase base no puede ser tratado como si fuera un objeto perteneciente a una clase derivada, pues no contiene todas las ranuras del objeto derivado.

- Herencia
  - Clase base
    - Referencia a una ranura de la clase base
  - Herencia sencilla
  - Conversiones de clase
  - Herencia múltiple
    - Ambigüedades
- Polimorfismo
  - Cómo trabaja el polimorfismo

Ambigüedades

Una clase derivada puede definir un elemento (ranura) del mismo nombre que una ranura en una clase base. A ésto se le llama *overriding* (dominación). Cuando se hace referencia al nombre dominado, el compilador supone que se desea accesar la ranura correspondiente a la clase derivada (a la más específica, a la mas cercana a las "hojas" y mas leja de la raíz del árbol de clases). De manera que si **r** es el nombre de la ranura duplicada, tenemos que dentro del alcance de la clase derivada, **r** representa la ranura de la clase derivada, pero es posible referirnos a la ranura de la clase base anteponiendo el nombre de la clase base con :: al nombre **r**.

Herencia múltiple

Una clase puede tener más de una clase base. Esto se conoce como herencia múltiple, puesto que la clase derivada está heredando de más de una clase base. En el ejemplo que sigue, Derived hereda de Base1 y de Base2.

```
class Derived: public Base1, private Base2
{
  public:
      void Print ()
      { printf ("b1 = %d and b2 = %d", b1.Get()); }
};
```

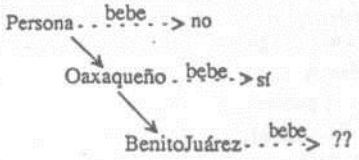
Si hay dos o más clases base con el mismo nombre para un método (o ranura), al referirnos a ella el compilador se quejará de una referencia ambigua, ya que carece de información suficiente para averiguar cuál de los dos métodos se heredará a la clase derivada.

```
Derived d;
d.Set(10);    //acá marca error el compilador pues no sabe si se refiere a Set de Base1[1] o a Set de Base2.
```

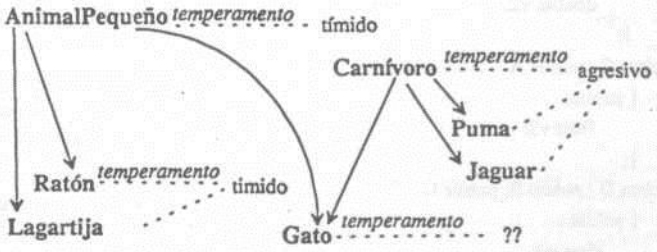
Para resolver la ambigüedad, el programador debe decir a cuál de los dos métodos se está refiriendo: por ejemplo, escribiendo d.base1::Set(10); o d.baset2::Set(10); segun sea su intención.

[1] Estamos suponiendo que las clases Base1 y Base2 han definido dos funciones distintas con el mismo nombre Set.

- **Ambigüedades**



- **Herencia múltiple**




```
class Derived: public Base1, private Base2
{
  public:
      void Print ()
      { printf ("b1 = %d and b2 = %d", b1.Get()); }
};

Derived d;
d.Set(10);
```

Clases base virtuales

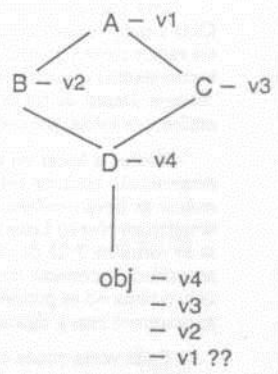
No se puede declarar la misma clase dos veces en la lista de clases base para una clase derivada. Sin embargo, es posible que la misma clase base aparezca mas de una vez en las clases ascendientes (progenitores) de una clase derivada. Esto generará errores, puesto que no hay manera de distinguir entre estas dos clases bases progenitoras.

```
class A
    { public:
        int v1;
     };
class B: public A
    { public:
        double v2;
    };
class C : public A
    { public:
        float v3;
    };
class D : public B, public C
    { public :
        char v4;
    };
int main ()
    { D obj :
      obj.v4 = 'c';
      obj.v3 = 17.25;
      obj.v2 = 3.3;
      obj.v1 = 30; }          //altamente ambiguo.
```

La solución a esto es declarar A como una clase base virtual tanto de de B como de C.

• Clases base virtuales

```
class A
   { public:
       int v1;
    };
class B: public A
   { public:
       double v2;
   };
class C : public A
   { public:
       float v3;
   };
class D : public B, public C
   { public :
      char v4;
   };
int main ()
   { D obj :
     obj.v4 = 'c';
     obj.v3 = 17.25;
     obj.v2 = 3.3;
     obj.v1 = 30; }          //altamente ambiguo
```



La solución a esto es declarar A como una clase base virtual tanto de de B como de C.

2.4 Ejemplo de una aplicación usando objetos

Una cadena comercial tiene tiendas en varias ciudades de varios estados de la República. Cada tienda vende varios ramos: blancos, enseres menores, ropa de hombre, ... Cada uno de estos ramos tiene a su vez divisiones más o menos elaboradas. Por ejemplo, en ropa de hombre tenemos camisas, pantalones, calcetines, ... Dentro de camisas tenemos: deportivas, para vestir, de trabajo. Denro de las de vestir vienen las de manga larga, manga corta. Luego por marcas, por estilos, por tallas, por colores.

Vamos a hacer un sistema de control de ventas con objetos. Su objetivo es poder saber en determinado instante (más o menos en tiempo real, es decir, "en el momento que yo quiera") cuánto se lleva vendido en cualquiera de las categorías anteriores. Por ejemplo, cuánto se lleva vendido en Nuevo León en todas las categorías; en Nuevo León en ropa de mujer; en Guadalajara en tornillos 7-25 de cabeza plana; en enseres menores en todo el país, etc. [En la práctica, la necesidad de conocer tan a detalle y "en el momento que sea" las ventas de un determinado artículo o rama no se presenta, y el control se hace digamos cada viernes, por ventas semanales. Pero igoremos esto y sigamos con el ejemplo.]

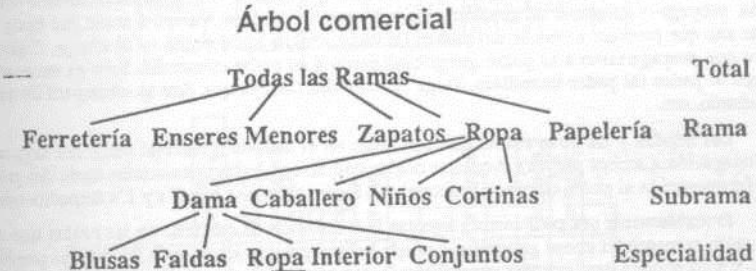
Cada venta puede clasificarse en dos grandes árboles: el árbol geográfico (dónde fué que se vendió) y el comercial (fue ferretería, artículos deportivos, ropa, ...). En un ejemplo real pueden intervenir otros árboles: el temporal (en qué hora, semana, mes, año) por ejemplo. Cada árbol tiene una subdivisión natural que cambia lentamente; por ejemplo, cada año o dos, conforme vayan apareciendo nuevos artículos en el mercado. Por esto conviene tener los árboles guardados en tablas o estructuras de datos, en vez de tenerlos "pre-alambrados" dentro del código del programa.

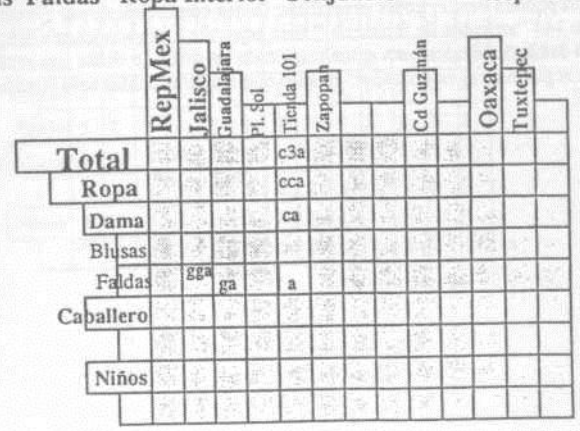
Podemos pensar en el "producto cartesiano" de estos árboles, que nos da una matriz (bidimensional en nuestro caso) donde cada celda tiene dos coordenadas, una geográfica y otra comercial. Por ejemplo, hay una celda representando "las ventas en Guadalajara de tornillos 7-25 de cabeza plana;" hay otra celda representando "las ventas en la tienda de Plaza del Sol en Guadalajara de tornillos 7-25 de cabeza plana," etc. Cada celda tiene exactamente un padre geográfico y otro padre comercial, excepto que algunas celdas solo tienen un padre y hay una celda (la de "ventas totales en todos los ramos en todo el pais") que no tiene padre alguno; es la raíz.

Vamos a usar cada celda como un acumulador, contador o "cubeta" donde las ventas se van guardando. Pero estos acumuladores también avisan o propagan a cada padre cuando su valor aumenta, pues el padre también debe ser incrementado. Y como el padre también tiene la propiedad de propagar este incremento a *su* padre, resulta que cada celda que debe ser incrementada (cuando una venta ocurre) lo será. De hecho, si no somos cuidadosos, algunas celdas podrían ser incrementadas doblemente, lo cual es un error.

En esta página solo hemos hecho una exposición del problema y un análisis conceptual de su representación y su posible solución. Todavía no hablamos de objetos. De hecho, este problema ha sido resuelto innumerables veces sin usar objetos, lo cual es perfectamente posible y quizá más rápido.


Árbol geográfico
País
República Mexicana
Ags BC BCSur Jalisco ... Zacatecas
Estado
Zapopan Cd. Guzmán Guadalajara Chapala
Ciudad
Tienda 101 Col. Libertad Tienda Plaza del Sol
Tienda


Árbol comercial
Total
Todas las Ramas
Ferretería Enseres Menores Zapatos Ropa Papelería
Rama
Dama Caballero Niños Cortinas
Subrama
Blusas Faldas Ropa Interior Conjuntos
Especialidad


RepMex
Jalisco
Guadalajara
Pl. Sol
Tienda 101
Zapopan
Cd Guzmán
Oaxaca
Tuxtepec
Total
Ropa
Dama
Blusas
Faldas
Caballero
Niños
c3a
cca
ca
a
gga
ga

La celda marcada con a representa las ventas de faldas en la tienda 101 de Guadalajara. Su padre geográfico es ga, que representa las ventas defaldas en Guadalajara. Su abuelo geográfico es la celda gga que representa las ventas de faldas en Jalisco.

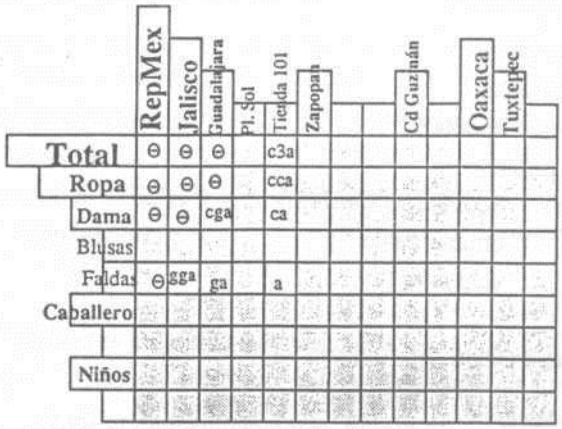
El papá comercial de a es la celda marcada ca, celda representando a las ventas de ropa de Dama en la tienda 101. Su abuelo comercial es cca, celda que acumula las ventas de ropa en la tienda 101. Su bisabuelo comercial es c^3a, que contiene el total de ventas en la tienda 101.

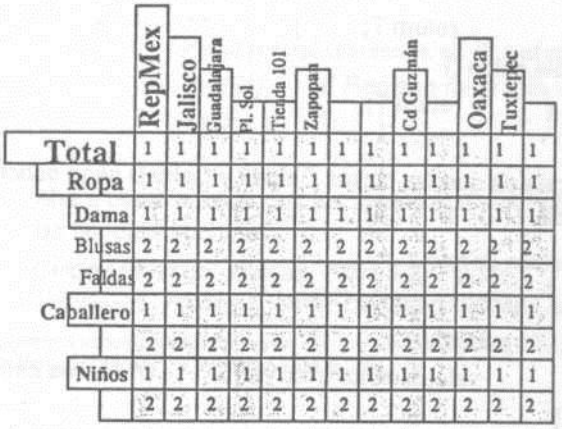
También un ascendiente de a (que carece de equivalente genealógico) es la celda cga, que representa las ventas en el papá geográfico de a, por el concepto "papá comercial de blusas"; es decir, las ventas en Guadalajara de ropa de dama. Hay otros ascendientes "mixtos" más, marcados con Θ.

La aplicación consiste de representar cada celda como un objeto que contiene un número (mis ventas, por lo que yo represento) y un método que cuando recibe un incremento o modificación, propaga o transmite tal modificación a su padre o padres. Vamos a tener dos tipos de celdas: uno que propaga a uno de sus padres (el comercial, o hacia arriba en el dibujo, digamos); el otro que propaga tanto a su padre geográfico como a su padre comercial. Solo es necesario propagar al padre (al padre inmediato, valga la redundancia), ya que *éste* se encargará de propagar al abuelo, etc.

Las tiendas y las otras celdas que son hojas en el sentido comercial serán del segundo tipo (propagación a ambos padres). Aquellas celdas que no son hojas comerciales serán del primer tipo (propagación al padre comercial solamente). Están marcadas con 2's y 1's respectivamente.

Probablemente sea conveniente agregar la restricción de que solo en las celdas que son hojas tanto comerciales como geográficas puede haber venta; es decir, solo las tiendas pueden vender, y solo pueden vender cosas específicas (hojas comerciales), por ejemplo, 144 tornillos 7-25, en vez de 144 "artículos de ferretería." Esto equivale a proporcionar a tales celdas (aquéllas que son hojas dobles) métodos para actualizar desde un menú (o desde una estación de punto de venta, o caja registradora) su contador, en anto que las otras celdas solo pueden actualizarse via propagación.


RepMex
Jalisco
Guadalajara
Pl. Sol
Tienda 101
Zapopan
Cd Guzmán
Oaxaca
Tuxtepec
Total
Ropa
Dama
Blusas
Faldas
Caballero
Niños


RepMex
Jalisco
Guadalajara
Pl. Sol
Tienda 101
Zapopan
Cd Guzmán
Oaxaca
Tuxtepec
Total
Ropa
Dama
Blusas
Faldas
Caballero
Niños

Código para celdas tipo 1

```
class Celda1
  {private:
        double T;                    /* cuánto he vendido */
   public:
        double incr (real val);      /* registra la nueva venta*/
        double saldo ();             /* cuánto tengo*/
        void print ();       }

double Celda::incr (double val)
   {        T := T + val;
             if (padre_comercial(myself) != NIL)
                    padre_comercial (myself)->incr(val);
            return T;       }
double saldo ()
   {        return T;     }
```

La idea es que las clases Celda2 y CeldaD sean subclases de Celda.

Código para celdas de tipo 2

```
class Celda2  :  public Celda1
  {    public:
          double incr (real val);        /* Nueva definición*/
  }

double Celda2::incr (double val)      /* redefino incr */
   {   Celda1::incr(val);             /* Uso la vieja definición*/
       padre_geográfico(myself).incr(val);
       return T;          }
```

• Código para celdas de tipo doble hoja

```
/* Son aquéllas celdas donde tiene sentido hacer una venta directa y no una "heredada." */
    Class CeldaD : public Celda2
                                    /* Dice que CeldaD es subclase de Celda2 */
    {public:
        int menú_de_venta (terminal ter)
                                    /* También cuenta con los métodos de Celda2*/
    }

    int menú_de_venta (terminal ter)
        /*hace una venta de este producto en esta tienda por la terminal ter*/
     {      int e=0;                    /* número de error */
            double x = 0.0;             /* total de la venta */
            ...
           /* aquí se despliegan menús, etc. El total regresa en x, digamos*/
            incr (x);     /* Registro mi propia venta x en T */
            return e;          }
```

Ahora declaramos todas las celdas que son hojas dobles, o sea, las tiendas-producto:

```
Celda2 PlazaSol_Blusas, PlazaSol_Faldas, ..., Tienda101_Blusas,
Tienda101_Faldas, Tienda101_Tornillos7x25, ....;
```

(el ejemplo supone irrealísticamente que hay un solo tipo de blusas color blanco, talla unitalla, etc., llamado "blusas", y lo mismo para faldas).

Luego declaramos las celdas de tipo 2 que no hayan sido ya declaradas como tiendas-producto:

(son aquéllas que tienen hoja comercial pero no tienen hoja geográfica)

```
Celda2 RepMex_Blusas, Jalisco_Blusas, Guadalajara_Blusas,
..., Oaxaca_Tornillos7x25, Tuxtepec_Tornillos7x25, ...;
```

(esta declaración tiene el mismo formato que int x, y, z;)

NOTA: Aquí se usa la clase Celda1 como tipo del argumento de estas funciones; en un llamado particular, el argumento puede ser de los tipos Celda1, Celda2 o CeldaD, puesto que los dos últimos son subclases (sub-tipos) del primero.

Algunos objetos no tienen padre comercial o padre geográfico; las rutinas padre_comercial y padre_geográfico regresan NIL en este caso, valor por el que se pregunta en la función incr antes de continuar la propagación.

Nota de eficiencia: La definición de **incr** propaga el incremento tan pronto como se ha efectuado; esto quiere decir que nuestro programa trata de mantener los valores de cada celda actualizados. Es un ejemplo de propagación hacia adelante o disparada por (la llegada de) datos.

Podríamos haber construido otra definición de **incr** donde se propagara hacia adelante un bit que indicase "contador obsoleto" (Además, si al propagarlo llegamos a un contador que ya está marcado como obsoleto, podemos terminar allí con confianza la propagación de este bit de obsolescencia). Esto evita propagar las nuevas ventas

Cuando vayamos a calcular el saldo con el método **saldo()**, las cosas ya no van a ser tan sencillas. Antes, como se tenía guardado el valor actualizado de T, la función saldo simplemente lo sacaba: return T. Ahora hay que ir a calcularlo: mi saldo es la suma de todos mis descendientes geográficos y comerciales, sin contar a ninguno dos veces. [El código de esta nueva función saldo no se exhibe.] Es un ejemplo de computación perezosa o disparada por la demanda. Una vez calculado, lo guardo en T y marco mi bit de obsolescencia como "actualizado." También, al ir a calcular los valores actualizados de mis descendientes, ellos mismos maracarán actualizado su total T. Nótese que si no uso el bit de obsolescencia, entonces cada vez que necesito un valor, necesito ir a calcularlo —una operación sin duda larga, que se podría evitar si el bit de obsolescencia existiese y nos dijera que el valor que reside en T es actual.—

En la práctica, no es necesario estar calculando en tiempo real cada vez que hay una venta, los subtotales de la tabla o matriz conceptual. Es mejor definirlos que se calculen bajo demanda, es decir, en el momento en que se necesitan. Probablemente sí sea aconsejable usar el bit de obsolescencia para evitar repetir cálculos tediosos. Nótese que C++ no proporciona una declaración de "calcúlese bajo demanda" o "calcúlese tan pronto se tengan datos." El programador tiene que escribir el código correspondiente. Empero, la fórmula que relaciona un totalizador con sus descendientes es la misma; por consiguiente, es posible en principio escribir un programa que genere código (a partir de tal fórmula) que sea de propagación ya bien sea disparada por datos o bajo demanda, según el programador lo indique. Esto es lo que sucede en CYC, por ejemplo.

Nótese también que bien se podrían tener unos totalizadores calculándose disparados por datos, y otros calculándose bajo demanda.

- Ahora declaramos las celdas que no son hojas comerciales:

  Celda1 RepMex_Total, Jalisco_Total, Guadalajara_Total, TiendaElSol_Total, Tienda101_Total, ..., RepMex_Ropa, Jalisco_Ropa, ..., Oaxaca_Niños, Tuxtepec_Niños, ...;

Ahora formemos la tabla o estructura que nos guarda los árboles comercial y geográfico:

Celda	Papá comercial	Papá geográfico
Tienda101_Blusas	Tienda101_Dama	Guadalajara_Blusas
TiendaElSol_Blusas	TiendaElSol_Dama	Guadalajara_Blusas
...		
Tienda101_Dama	Tienda101_Ropa	Guadalajara_Dama
...		
Guadalajara_Dama	Guadalajara_Ropa	Jalisco_Dama
Jalisco_Dama	Jalisco_Ropa	RepMex_Dama
Jalisco_Ropa	Jalisco_Total	RepMex_Ropa
Jalisco_Total	NIL	RepMex_Total
RepMex_Ropa	RepMex_Total	NIL
RepMex_Total	NIL	NIL

Por último definamos las rutinas padre_comercial (Celda1) y padre_geográfico (Celda1) como búsquedas en la tabla anterior.
Usan un argumento que es un objeto de clase Celda1(pero bien puede ser de Celda2 o CeldaD).

2.5 Lenguajes que manejan objetos

Como los objetos ahora están de moda, resulta que todo mundo dice, con cierta razón, que su lenguaje es "orientado" a objetos. Por esto conviene darle un adjetivo más fuerte o claro a los lenguajes que *claramente* usan objetos, y quitarles lo de "orientado." De manera que los siguientes son lenguajes que *manejan* objetos:

Smalltalk. El original. El inventor de los objetos.

CLOS. Common Lisp Oriented System. CLOS es el lenguaje Lisp con objetos.

C++. Una extension al lenguaje C, que maneja objetos. El más popular. Hay varias variantes o marcas:

Gnu C++. Popular, bueno y gratis. Obténgase en The Free Software Foundation. 675 Massachusetts Ave. Cambridge, MA. 02139.

Borland C++. Para PC's.

AT&T C++. El oficial.

Glockenspiel C++. Casi igual al Gnu C++. Lo usa la base de objetos Ontologics.

Saber C++. Centerline. 185 Alewife Brook Parkway. Cambridge, MA. 02138. Tel (617) 876 7636. C++ con una interfaz gráfica.

Objective C. Otra variante de C que tambien maneja objetos. No es compatible con el C++; es similar. No es tan popular como el C++. Lo usa la máquina NeXT.

Classic Ada. Agregado a Ada para hacerlo orientado a objetos. Software Productivity Solutions. Melbourne, Fla.

Eiffel. Nacido en un ambiente de investigación.

Linda. Nacido en un ambiente de investigación.

- Lenguajes para objetos
  - Smalltalk
  - CLOS
  - C++
    - Gnu C++ — Pre-procesador a C
    - Turbo C++ (para PC's) — Compilador
    - Zortech C++ (para PC's) — Compilador
    - AT&T C++ [para Unix; también hay para PC's] — Pre-proc
    - Glockenspiel C++ — Pre-proc
  - Objective C
  - Classic Ada
  - Eiffel
- Linda

2.6 Dando permanencia a los objetos

Hay varias maneras de darle permanencia a los objetos.

No guardar objetos permanentemente; crearlos cada vez

Si se usan solo unos cuantos objetos en memoria, no habrá necesidad de guardarlos en disco (en forma permanente, segun se vió en la seccion 2.7); bastara con crearlos de nuevo cada vez que el programa se ejecuta. Es decir, estos objetos solo son residentes en memoria. *Esto es lo que ofrece C++: objetos residentes en memoria solamente*. Si uno desea hacerlos permanente, es problema de uno.

2.6.1 Guardando objetos en archivos planos

Si se desea guardar los objetos, podemos guardarlos primeramente en archivos ascii, te niendo cuidado de guardar todos los objetos que estén en memoria, a fin de no dejar apuntadores sin resolver. Hay que utilizar el procedimiento de conversión de apuntadores locales que se h descrito en la sección 2.7. Es decir, para cada clase (*pero en realidad no es para cada clase Usar herencia!*) hay que escribir una función o método **guarda_a_disco** que los exporta a me moria permanente. Como también tendremos que leerlos después de ascii a memoria, habrá qu escribir otro método para cada clase, **lee_a_memoria**.

Guardando objetos en bases de datos. Ver sección 2.6.2

Guardando objetos en bases de objetos. Ver sección 2.6.3

Mezclando bases de datos con bases de objetos

La mezcla de bases de objetos con bases de datos relacionales debe verse con escepticism la razán es que las bases de objetos hacen todo lo que las bases relacionales hacen, y más. P consiguiente, si se necesita el poderío de una base de objetos, no tiene caso usar también bas de datos relacionales (a menos que se haya efectuado un parche, que usa objetos, a un progra anterior que estaba accesando una base de datos relacional). En general, si se está a gusto c una base de datos relacional, no hay razón para pasarnos a una base de objetos, con su nomencl tura y semántica nueva, y con su inmadurez relativa (por ser productos recientes).

- Dónde guardar los objetos que están en memoria?
  - No los guarde. Créelos de nuevo cada vez
  - Guarde los datos necesarios para su creación en algun archivo plano
    - Guárdelos en archivos planos ascii
    - Transformar los apuntadores locales a apuntadores ascii
    - Para cada clase
      * método *guarda_a_disco* (o exportador); método *lee_a_memoria* (o importador)
      * En realidad, usando herencia, el trabajo se simplifica

      Barriendo la tabla de objetos, vaciar todos a memoria
- Usando objetos con bases de datos relacionales
  - A primera vista no conviene mezclar estos bases de objetos con bases de datos relacionales
  - Empero, a veces conviene guardar objetos (es decir, mezclar objetos, pero no *bases de objetos*) con bases de datos relacionales.
  - Si se necesita el poderío de una base de objetos, y está uno dispuesto a pagar el "alto precio", no hay ventaja alguna en usar bd relacionales
  - Excepción: Cuando por razones históricas hay que accesar una bd relacional

2.6.2 Cómo almacenar objetos en bases de datos

Un poco más sofisticado que guardar los objetos en archivos planos, es guardarlos en bas de datos relacionales. Los mismos principios se observan. Cada apuntador a objeto hay que tr ducirlo a un "apuntador ascii" que puede ser simplemente el nombre del objeto, siempre y cua do sea único (por cierto, si se decide tener objetos con nombres únicos, la rutina de lectura nombres debe protestar si ya existe otro objeto con el mismo nombre, y exigir que se le de nombre distinto).

Cada ranura puede guardarse como una *relación* (relación de base de datos) entre dos obj tos; por ejemplo, la ranura (Guzmán edad 40) puede guardarse en la base de datos como una bla llamada **edad** que tiene dos campos o columnas, uno con los nombres de los objetos, el o con las edades.

De la misma manera, una ranura con valores múltiples, por ejemplo, la ranura (Guzm amigo Juan Pedro JoséLuis María) puede representarse por las relaciones binarias (Guzmán an go Juan), (Guzmán amigo Pedro), (Guzmán amigo JoséLuis), y (Guzmán amigo María), las q se guardan en una tabla llamada **amigo** con dos columnas, ambas de personas.

ChenHo [41] explica con algunos detalles el método.

Para accesar

En objetos, la edad de Guzmán se obtiene simplemente por Guzmán.edad; en la base de tos relacional, habrá que ir a la tabla edad y buscar por la "llave" Guzmán (la cual es única). simplifica el problema si todas las ranuras tienen un nombre único; en este caso, la tabla edad la única que hay. Pero supongamos que hay dos conceptos con el mismo nombre, como r (parte de la planta que obtiene nutrientes del suelo) y raíz (valor que anula a un polinomio; ab sa donde la gráfica del polinomio cruza el eje de las x's). Si no deseamos que el usuario distir entre estos dos conceptos o acepciones del vocablo raíz, forzándolo a que diga, por ejemp raíz-vegetal y raíz-matemática, entonces le permitiremos que use raíz, y por el contexto (po tipo de objeto que puede poseer raíz) observaremos de que raíz se trata.

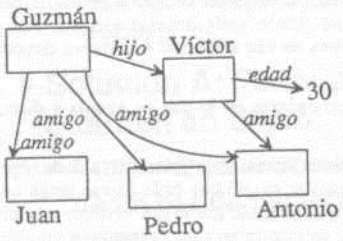
Para los efectos de guardar en la base de datos relacionales, habrá que tener dos tablas, "externamente" se llaman raíz pero que internamente se denominan por ejemplo raíz1 y raíz2 mejor aun, raíz-vegetal y raíz-matemática. Cuando se nos pregunte el valor de la raíz de un ob to, viendo el tipo de objeto (¿es planta o es polinomio?) podremos ir a la tabla adecuada.

Para accesar la edad del hijo de Guzmán, en objetos se escribe Guzmán.hijo.edad. El ope dor (.) es asociativo a la izquierda, puesto que primero se evalúa Guzmán.hijo, dando el obj Víctor; Víctor.edad da como resultado 2.

Guardando los objetos en bases de datos relacionales

- Cada objeto se hace único y se convierte en un nombre (ascii)
- Cada ranura se convierte en una tabla o relación en la base de datos
- Una ranura multi-valuada produce varias entradas (varios renglones) en la relación que lleva su nombre

Objetos



B. de datos relacional

amigo

Guzmán	Juan
Guzmán	Pedro
Guzmán	Antonio
Víctor	Antonio

edad

Víctor	30

hijo

Guzmán	Víctor

2.6.3. Almacenando objetos en bases de objetos. Sistemas comerciales

Las bases de objetos (Itasca[38], antes Orion; Versan [65]; Object Store [52]) se inventaron para darles permanencia a los objetos; es decir, para guardarlos permanentemente (en disco). Esto es porque en la práctica es importante guardar los objetos con los que se ha estado trabajando, de manera que sobrevivan de una corrida del programa a la siguiente.

El problema principal que tienen que resolver las bases de objetos es de como guardar los valores guardados en cada ranura, y también el problema de guardar los metodos. En memoria principal, las ranuras contienen dos tipos de datos: (a) los datos primitivos (cadenas, enteros, números reales) se guardan "ellos mismos" en la ranura, es decir, no se guarda un apuntador a ellos (una posible excepción son las cadenas); (b) en cambio, cuando una ranura guarda (tiene como valor) a un objeto o a un conjunto de ellos, este valor se guarda como un apuntador al lugar de la memoria donde está almacenado el objeto.

Es decir, cada objeto (pero no cada dato primitivo) se guarda en un lugar único en memoria, y se puede hacer referencia a tal objeto por medio de un apuntador al mismo. Desde este punto de vista, los objetos no son más que un identificador (el nombre del objeto) mas un manojo de ranuras, las que contienen ya sea datos primitivos (cadenas, enteros, ...) o apuntadores a otros objetos. Como tal, estas estructuras parecería que fuera facil copiarlas tal cual a disco, y traerlas despues cuando se les necesite nuevamente.

El problema está en que esos objetos, al ser leídos nuevamente (en alguna otra corrida del programa, o en otro programa) del disco a la memoria, no van a ser cargados necesariamente en la misma dirección del espacio de direccionamiento donde antiguamente estaban. Por consiguiente, los apuntadores van a volverse inválidos, pues no van a apuntar a la nueva dirección de carga de cada objeto, sino a la direccion antigua.

La solución en general consiste en no usar apuntadores en lo que se guarda a disco. Hay cuando menos dos soluciones posibles:

(1) Convertir los apuntadores (que son direcciones de memoria principal) a cada objeto por el nombre en ascii del objeto apuntado, si previamente se exigió que cada objeto tenga un nombre único. La cadena en ascii (el nombre del objeto) se precede por algun símbolo especial que denote "yo no soy realmente una cadena cualquiera, yo substituyo a un apuntador a un objeto cuyo nombre ahora poseo." Cada ranura de cada objeto, al guardarse en disco, convierte su valor como sigue: los apuntadores a objetos como ya se describió; los números, cadenas y otros objetos primitivos, a notación ascii convencional. Al leerse a memoria, los objetos primitivos pasaran a ocupar directamente su lugar en la ranura, en tanto que los "apuntadores ascii a objetos" seran convertidos a una direccion de memoria donde el referido objeto (cuyo nombre lleva) esté guardado, guardándolo si es necesario, en caso de que no se encuentre. Por este motivo, todos los objetos que vienen de (o van a) disco pasan por una tabla de dispersión (tabla hash) o tabla de objetos, que convierte apuntadores a objetos (dirección en memoria principal donde se encuentra el objeto depositado) a cadenas ascii (nombre del objeto)

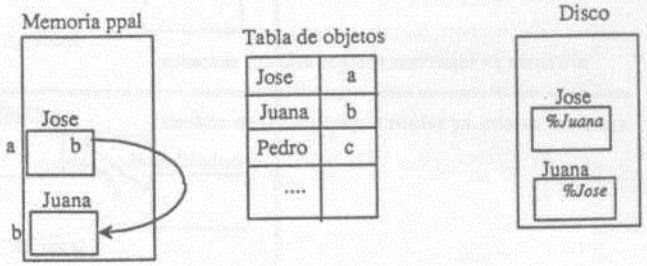
- **Bases de objetos**
  - **Objetivo: darle permanencia a los objetos (guardándolos en disco, por ejemplo)**
  - **Problema: en memoria, las ranuras contienen apuntadores a objetos**

    Cada objeto tiene un lugar úen memoria. esto permite manejar apuntadores a los objetos, simplemente

    Estos apuntadores son válidos solo dentro del espacio de direccionamiento actual. **Carecerán de sentido en el disco**
  - **La tabla de objetos**

    * Conoce la posición en memoria de cada objeto. Asocia el nombre del objeto (cadena) con el objeto (apuntador único)

    * Se usará después par sber si un objeto existe pero no está en memoria
  - **Solucion A: Pasar a ascii los apuntadores que se guarden en disco**
    - **Contendrán el nombre del objeto apuntado, adornado de un par de caracteres ascii que significan "yo soy objeto"**



(2) La otra solución consiste en usar un espacio de direccionamiento independiente del de la memoria principal (es decir, las direcciones en este espacio no son idénticas con las de la memoria principal, pero hay una correspondencia uno a uno entre ellas). Cada apuntador a memoria principal, llamado apuntador local, se convierte a un apuntador "independiente", llamado apuntador global antes de ser almacenado en disco. De nuevo, se usa una tabla de objetos para hacer esta conversión.

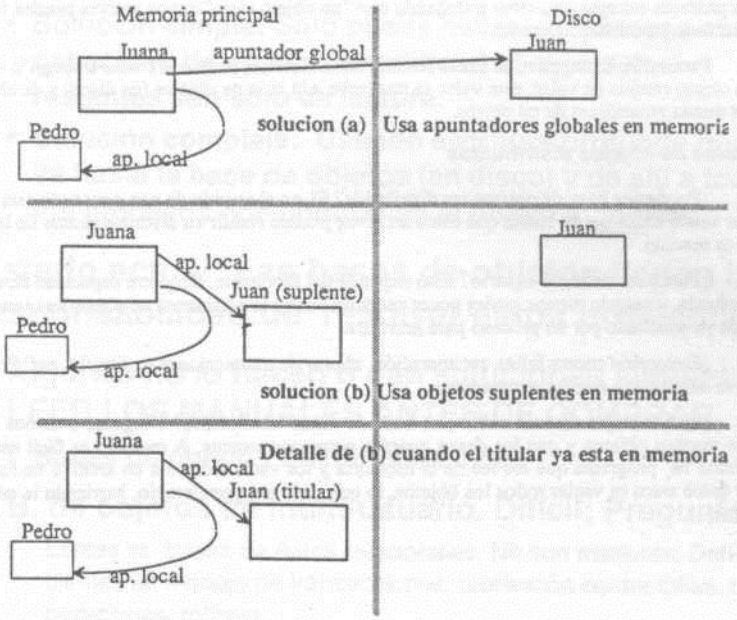
La razón de llamarse "apuntador local" es que se refiere solo a los objetos que están en memoria principal. En aplicaciones que tienen muchos objetos, todos ellos se encuentran en la base de objetos (en disco), en tanto que solo una parte de ellos se trae a memorial. Por este motivo también, se les llama apuntadores globales a aquellos que se encuentran almacenados en disco.

La conversión de apuntadores locales a globales (o a apuntadores ascii) la realizan de manera automática las bases de objetos.

Interacción entre los objetos permanentes y aquéllos en memoria

Si no se han traido todos los objetos a memoria, es posible que algunos de los ya traidos apunten a otros que aun estan en disco. Es decir, es posible que contengan apuntadores globales. Debe haber, pues, una manera de identificar si un apuntador es local o global, si se desea permitir que solo parte de los objetos se bajen a memoria de disco. Una manera es usar apuntadores grandes para los apuntadores globales, y menos bits para los locales. Otra manera, utilizada por Ontos [54] es que los objetos globales apuntados desde memoria, se substituyen (en memoria) por *suplentes*, que son objetos "flacos" porque sus ranuras carecen de valor (ya que estos no se han leido de memoria). Cuando se desea accesar un objeto y en vez de él se accesa al suplente, éste se encarga de traer a su titular (es decir, la información de sus ranuras) de disco. El objeto suplente apuntara entonces a su titular; apuntaba a NIL cuando este no estaba en memoria. Esta interacción entre objetos en memoria y en disco la realizan automáticamente las bases de objetos.

- **Solución B:** usar apuntadores globales en disco
  - Los apuntadores en memoria se llaman "locales"
- **Interacción entre los objetos permanentes y aquéllos en memoria**
  - a. Conservando en memoria los apuntadores globales
  - b. Usando objetos suplentes (*surrogates*) para aquellos objetos que no han sido traido aun a memoria
  - Cuando se trae a memoria, el suplente apunta al titular



Compartiendo una misma base de objetos entre diversos procesos

Con las bases de objetos es posible que varios procesos (cada uno con su espacio de direccionamiento) compartan objetos de una misma base de objetos. En primer lugar, si los procesos ni se comunican entre ellos ni comparten algunos objetos comunes , esto significa que no hay interacción alguna entre tales procesos, por lo que cada uno procede independientemente con respecto a la actualización y uso de la base de objetos.

Ahora bien, supongamos que los procesos se comunican entre ellos o comparten ciertos objetos. Si un proceso le va a mandar un objeto a otro (o se va a referir a algun objeto, como argumento de una función para que el otro proceso la ejecute, digamos), el remitente debe convertir su apuntador (local) a un apuntador global (o a un nombre ascii) para que el objeto destinatario le entienda y pueda descifrar de qué objeto se trata, ya que los apuntadores locales no son entendibles (no tienen sentido) para otro proceso que no sea el local. Solo a través de los apuntadores globales (o de los apuntadores en ascii) se pueden pasar objetos estos procesos.

Cuando se está compartiendo el mismo objeto por varios procesos, existe el riesgo de que los cambios de uno no los vea el otro. Las bases de objetos por lo general se van por el lado fácil . Cuando un proceso extrae ciertos objetos de la base de objetos (en disco), ningun otro proceso los puede modificar. Están "prestados" al primero que los "saco" de disco a memoria. Como solo un proceso tiene la version actual del objeto (los otros tienen copias viejas solo de lectura), todos los procesos excepto uno estan trabajando con "un objeto viejo" cuyos valores pueden haber cambiado ya substancialmente:

Para evitar lo anterior, se usa la técnica de la escritura profunda (write-through[2]); cuando un objeto cambia de valor, este valor se transmite a la base de objetos (en disco) y de ahí a todos los demas poseedores de tal objeto.

Bases de objetos distribuidas

¿Puede una base de objetos ser distribuida? Sí, en el sentido de que a menudo esta formada por varios archivos, de suerte que estos archivos pueden residir en distintos puntos de la red. Esto es sencillo.

¿Puede ser de multi-usuario? Esto depende del fabricante. Requiere capacidad de escritura profunda, o cuando menos, poder poner candado contra la escritura a un objeto una vez que ha sido ya solicitado por un proceso para escritura.

¿Protección contra fallas, recuperación, aborto de transacciones (rollback). etc? Pocas. Las b. de objetos son nuevas, no maduras.

¿Qué ventajas ofrecen sobre una base de datos relacionales? Ninguna, a menos que usted use muchos objetos y que los desee guardar permanentemente. A menudo es fácil escribir uno mismo un programa que los lee de la memoria y los vacía a disco a un archivo en forma ascii. El único truco es vaciar todos los objetos, lo que se logra, por ejemplo, barriendo la tabla de objetos.

---

[2] Misma que se usa para escribir a través de la memoria caché directamente a una memoria RAM compartida

- Compartiendo una base de objetos entre diversos procesos
  - A. Ningún proceso se comunica ni comparte objetos con otro
    - Caso simple. Cada proceso es independiente
  - B1. Un proceso envía objetos a otro (a través de mensajes)
    - Los apuntadores locales deben globalizarse primero
  - B2. Varios procesos comparten (tienen en memoria) un mismo objeto
    - Solución simple: Solo puede haber un proceso con capacidad de escribir (modificar) el objeto, los restantes son solo de lectura.
    - Solución compleja: Usando escritura profunda que va hasta la base de objetos (en disco) y de ahi a todos los procesos que comparten el objeto modificado
- Estado actual: Las bases de objetos tienen la responsabilidad de resolver estos problemas
  - Algunas no lo hacen o dan soluciones simples. LEER LOS MANUALES ANTES DE COMPRAR
    - Bases de objetos distribuidas. Fácil
  - B. de objetos de multi-usuario. Difícil; Preguntar.

    Lentas vs. bases de datos relacionales. No son maduras. Deficientes en manejo de transacciones, protección contra fallas, recuperaciones, rollback.

    NO SE USEN bases de objetos a menos que no haya otra solución

Ref. Críticas de C. J. Date
pags 90-95

Bases de objetos comerciales

Son parientes de las bases de datos. Guardan en memoria secundaria (en disco) objetos. Son útiles cuando se necesitan manejar grandes conjuntos de tipos complejos de datos y estructuras, tales como video y sonido. Se acomodan bien para aplicaciones tales como programas de grupo, multi medios y Diseño ayudado por Computadora.

O2. O2 Technology. Versalles, Francia. Para Unix.

Itasca. Istasca Systems. 2850 Metro Dr., Suite 300. Bloomington, MN 55425 Tel. (612) 851 3153. Es la versión comercial del sistema Orion, construido en MCC. Para Unix y máquinas Symbolics de Lisp.

Object Store. Object Design, Inc. One England Executive Park. Burlington, MA. 01803. Tel. (617) 270 9797. Unix, Windows.

Objectivity/DB. Objectivity Inc. Menlo Park, CA. Para Unix.

Open OODB. HP.

Versant. Versant Object Technology. 4500 Bohannon Dr., Ste 200. Menlo Park, CA. 94205 Tel. (415) 325 2300. Para Unix.

Ontos. Ontologics, Inc. Burlington, MA. Guarda objetos de C++. Para Unix. Para PC-Unix SCO; OS/2. Ψ[3] La versión de Unix para Sun es bastante lenta. Ψ Además, cuando se usa varias veces, se va comiendo la memoria disponible (parece que no hace recolección de basura correctamente).

Graphael. Francia.

Gemstone. Servio Corp. Alameda, CA. Unix; además, admite clientes en PC y en MacIntosh.

[3] Ψ denota a su derecha un comentario de usuario (no del proveedor).

- Bases de objetos comerciales
  - O2
  - Itasca
  - Object Store
  - Open OODB. HP
  - Objectivity
  - Versant
  - Ontos
  - Graphael
- Gemstone

Vista global del curso

Curso "Construcción de Software de Calidad"

I Diseño con objetos

- Diseño estructural con objetos
- Diseño con responsabilidades

II Otros métodos de diseño

- Componentes rígidos y flexibles

III Prototipos rápidos

IV Estrategias de implementación

V Conexión con sistemas ya existentes

VI Paquetización

VII Calidad del Software

VIII Pruebas, documentación

IX Expectativas de las herramientas CASE

- **Terminología**
  - **Objeto**

    Una abstracción de una entidad del mundo real. Representados como estructuras de datos (variables de instancia, o ranuras) + métodos (programas que las manipulan).

    * Propiedad, ranura o relación * Método

    Una propiedad puede verse como una relación lógica
    - **Clase**

      Un conjunto de objetos que tienen características comunes
    - **Individuo**

      Un objeto que no es una clase. "Elemento" o miembro de una clase (vista como un conjunto de elementos).
    - **Método**

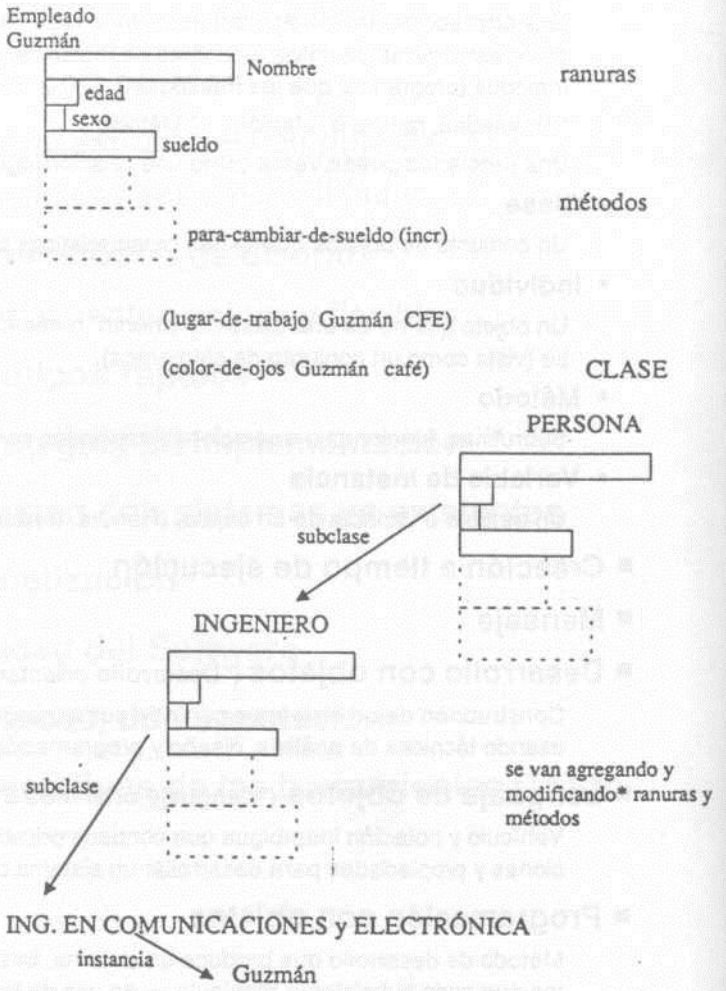
      Subrutinas, funciones u operaciones asociadas con un objeto
    - **Variable de instancia**

      Un atributo o variable de un objeto. (Ranura, o relación)
  - **Creación a tiempo de ejecución**
  - **Mensaje**
  - **Desarrollo con objetos** ("Desarrollo orientado a objetos")

    Construcción de un sistema a partir de sus especificaciones, usando técnicas de análisis, diseño y programación con objetos
  - **Lenguaje de objetos** ("Lenguaje orientado a objetos")

    Vehículo y notación inambigua que contiene primitivas, especificaciones y propiedades para desarrollar un sistema con objetos
  - **Programación con objetos**

    Método de desarrollo que produce un sistema basado en los objetos que cada subsistema manipula — en vez de basarse en las funciones o transformaciones que el sistema debe realizar


Empleado
Guzmán
Nombre
edad
sexo
sueldo
ranuras
métodos
para-cambiar-de-sueldo (incr)
(lugar-de-trabajo Guzmán CFE)
(color-de-ojos Guzmán café)
CLASE
PERSONA
subclase
INGENIERO
subclase
se van agregando y modificando* ranuras y métodos
ING. EN COMUNICACIONES y ELECTRÓNICA
instancia
Guzmán

• Características de los objetos

- **Ocultamiento de la información (encapsulamiento)**

  Separación de los detalles de representación (o implementación) de un objeto, clase o sistema de sus detalles relevantes al dominio de la aplicación.

  Los programas que usan objetos no deben saber de éstos más de lo necesario. Más de lo que los objetos mismos hacen público.

- **Abstracción**

  Separación de detalles innecesarios en los requerimientos o especificaciones de un sistema, para facilitar la comprensión de tales requerimientos o especificaciones

- **Liga dinámica**

  Instanciamiento o creación de un identificador, variable o método cuando se va a usar — es decir, durante la ejecución.

- **Herencia**

  Relación entre dos clases de manera que la subclase toma todas las características (métodos y variables) de la superclase.

- **Polimorfismo (sobrecarga de funciones)**

  Existen nombres genéricos de funciones (métodos), los que pueden ser reemplazados en las subclases por otras funciones más específicas o ligeramente diferentes *con los mismos nombres.* Ejemplo: "+" en Fortran para sumar reales o enteros.

  Este uso de un mismo nombre para representar funciones similares es completamente intencional. El método específico correspondiente lo escoge:

  * El compilador (liga estática) a tiempo de compilación; o

  * El despachador de métodos (liga dinámica) a tiempo de ejecución. Ejemplo: Método *saldo* para el objeto *cuenta-bancaria.*

- **Ranuras o propiedades**

  "Slots", variables de instancia. relaciones (no confundir con las relaciones del método E-R-A, a las que se parecen).

  - **Inversas**
  - **Verificación automática del tipo (clase) de un valor**
  - **Representación en memoria**
  - **Propiedades de las ranuras**
    - **¿Son también objetos?**
      * No en C++
      * Sí en CLOS.
    - **Dominio**

      La clase más general para la cual pueden estar definidas

      Ejemplo: edad. Su dominio es Persona. o Animal.
    - **Rango**

      La clase más general de la cual pueden tener individuos (elementos) como valor.

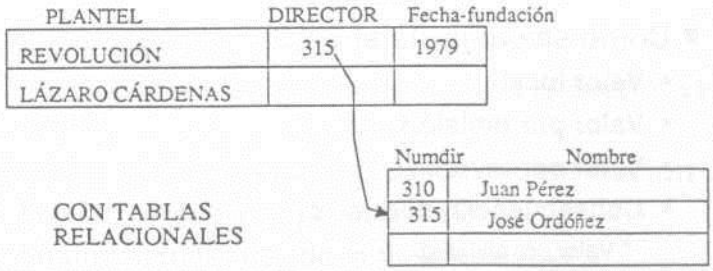
      Ejemplo: edad. Su rango es Entero. o Número (demasiado general). Número no negativo (bien).
    - **Cardinalidad**

      O Aridad. Número de valores que la ranura puede tener.

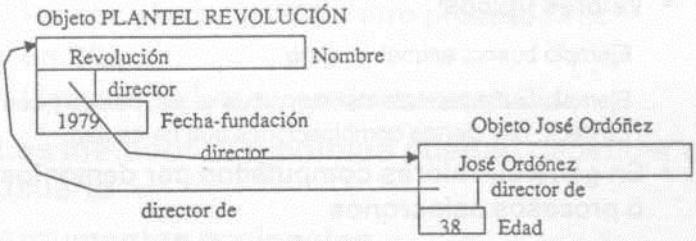
      Ranuras con un solo valor: RFC, longitud, padre.

      Ranuras con varios valores: amigo, hijo, cuentas-bancarias.
    - **Soporte**

      Conjunto de valores (relaciones) y reglas (demonios) que afirman que esta ranura tiene este valor

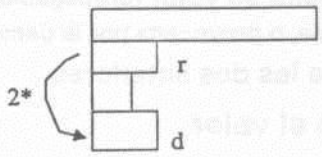


CON TABLAS RELACIONALES



CON OBJETOS

USO DE RANURAS INVERSAS EN LOS OBJETOS



RELACIONES ENTRE RANURAS

- Valores de una propiedad o ranura
  - Cómo se computa el valor
    - Valor local
    - Valor por omisión
    - Valor por inversa
    - Consecuencia, deducción
      * Valor por afinidad
    - Valor por tarea o agenda
    - Valores típicos

      Ejemplo bueno: animal ⇨ gato

      Ejemplo malo: animal ⇨ ser vivo que vuela, peludo y con agallas. Es decir, hay ciertas combinaciones que no se dan
    - En general, valores computados por demonios, reglas o procesos asíncronos
  - Cuándo se computa el valor
    - Cuando se guarda o deposita
    - Cuando se cambian los valores de otras ranuras de las cuales depende (propagación hacia delante, o provocada por los datos — "Data driven")
    - Cuando se necesita su valor (propagación hacia atrás, evaluación perezosa, o provocada por la demanda)
    - Diferencias entre las dos anteriores
  - Qué sucede con el valor
    - Se guarda una vez conocido. "Caching". Ventaja: rápido

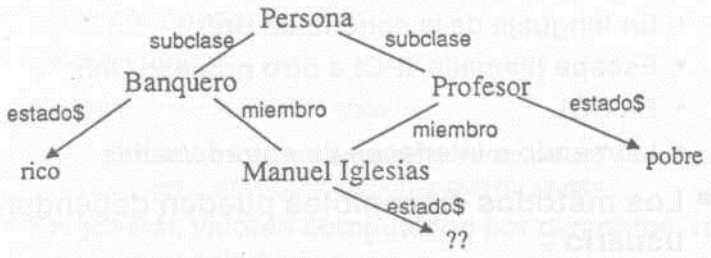
      Se guarda en la ranura. Para eso es.
    - Se descarta. Se vuelve a calcular cada vez que se necesita

      Ventaja: siempre está actualizado (para computación distribuida).

- Métodos. Cómo se definen
  - Programa en C++

    Programas en el lenguaje donde los objetos se definen. Es lo más común.
  - Escape a otros paradigmas de programación
    - En otro lenguaje de programación
    - En un lenguaje 4GL de la base de datos
    - En lenguaje de la concha de Unix
    - Escape (llamada, IPC) a otro proceso Unix
    - En SQL
    - Llamando a interfaces de entrada/salida
  - Los métodos disponibles pueden depender del usuario
  - Argumentos opcionales
  - Número variable de argumentos

- Herencia
  - Qué se hereda
  - Herencia simple
  - Herencia múltiple
    - Conflictos al heredar valores por omisión
    - Conflictos al heredar métodos



  - Herencia a través de cualquier ranura
- Cuándo se hereda
  - En el momento de producir el nuevo objeto
  - En el momento de dar un valor local
  - Las ranuras sin valores, ¿se heredan?
    - Sí, lógicamente

      Todo objeto "tiene derecho" a tener las ranuras definidas en su clase.
    - No, físicamente

      En algunos lenguajes se construyen las ranuras solo cuando se necesitan. Detalle de implementación.

      En C++ se construyen todas las ranuras al crear el objeto

- Propiedades inversas
  - Qué son
    - Ventajas
    - Desventajas
    - La inversa está en "el otro" objeto
  - En qué casos conviene no tenerlas
  - La inversa debe ser única
    - La inversa de la inversa de r es r
  - Modificando una ranura se modifica su inversa, y viceversa
    - Esta modificación debe ser automática — ¡es la ventaja de la inversa! Pero no lo es en C++.

      Hay que programar a mano el método de ir a modificar el valor de la ranura inversa

- Propiedades derivadas o computadas
  - Cuándo conviene definirlas
  - Cómo se implementan
    - Con demonios o procesos asincrónico
      - Los demonios están definidos junto a las ranuras que modifican
      - Mediante fórmulas que describen o definen a los demonios
    - Computadas por programas en otro lenguaje de programación
    - Accesando una base de datos
      - Por ejemplo, usando SQL.
      - O usando el lenguaje de la base de datos
- Recordando los valores de las ranuras
  - Cacheo ("caching")

    Ranuras que se reflejan en memoria. Son las más comunes
  - Ranuras que se reflejan en disco

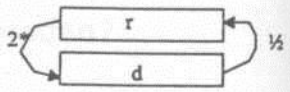
    Necesarias para compartir datos en computación distribuida
- Propagación de valores
  - El problema de los errores de redondeo

    $d = 2\,r$

    $r = d / 2$

    $r = 3$ de donde $d = 6$

    ¿o no?

  - El problema de ciclos inherentes

    Si tienes una silla verde, píntala de rojo

    Si tienes una silla roja, píntala de verde

- **Diseño estructural orientado a objetos ("OOSD")**

  Wasserman, Pircher, Muller. "An object-oriented structured design method for code generation," *Software Eng. Notes*, Vol. 14, No. 1, Jan 89, pp 32-55.

  - **Notación para describir un diseño arquitectónico**

    Es un diseño de alto nivel que identifica módulos individuales pero no su representación interna detallada.

  - **Usa un diagrama de estructuras orientado a objetos**

    - **Similar a los diagramas estructurales comunes**

      * Contienen módulos, parámetros de datos, parámetros de control
      * Se le agregan notaciones para objetos, clases, métodos, instancias, herencia.
      * Se pueden usar para mostrar herencia múltiple, paso de mensajes, polimorfismo, ligas dinámicas
      * También apoya el uso de monitores

- **Diseño orientado a objetos (Booch) ("OOD")**

  G. Booch. "What is and what isn't object-oriented design?" *Am. Programmer*, Vol. 2, No. 7-8, Summer 1989, pp. 14-21.

  - **El más popular de los métodos ("metodologías") de diseño con objetos**
  - **Es una alternativa y no una extensión al diseño estructurado**
  - **No hay un orden específico en las fases de este método**
    - **Los analistas deben trabajar iterativa e incrementalmente**
  - **Cuatro pasos principales:**
    - **Identificar clases de objetos**

      Identificar las abstracciones principales ⇨ son buenos candidatos a clases
    - **Identificar la semántica de las clases**
      * Qué significan tales clases
      * Qué implica ser un miembro de tal clase
      * Usar escenificaciones ("scripts") para definir el ciclo de vida de cada objeto de su creación a su destrucción.
    - **Identificar las relaciones entre clases y entre objetos**
      * Patrones de herencia entre clases
      * Patrones de cooperación entre objetos
    - **Implementar las clases**
      * Construir vistas detalladas de clases y objetos
      * Definir sus servicios (métodos)
      * Asignar métodos a procesadores (en computación distribuida)

- Herramientas para el diseño orientado a objetos de Booch
  - Diagrama y esqueleto de clase
    - Enfatiza definiciones de clase y herencia

      Las clases se definen en nubes dibujadas con líneas punteadas

      Las relaciones se muestran con arcos dirigidos de diferentes tipos: usa, miembro de, herencia, metaclase, indefinida.
  - Diagrama y esqueleto de módulos
    - Hincapié en definiciones de mensajes, visibilidad, y control. Se usa cuando el lenguaje (Ada) soporta módulos
  - Diagrama y esqueleto de objeto
    - Definiciones de mensaje, visibilidad, control

      Cada objeto (en una nube punteada) representa un miembro arbitrario de una clase. Conectados por arcos dirigidos que definen la visibilidad de los objetos y los mensajes. *No muestra flujo de control *No muestra orden en los eventos
  - Esqueleto de operación (método)
    - Captura la definición de servicios. Texto o pseudo-código
  - Diagrama y esqueleto de proceso
    - Asigna módulos a procesadores
  - Diagrama de transición de estados
    - Modela los estados de un objeto y las transiciones entre ellos
  - Diagrama de tiempos
    - Compañero del diagrama de objeto

      Muestra el flujo de control y orden de eventos entre un grupo de objetos que colaboran

• Diseño basado en responsabilidades

Wirfs-Brock, Wilkerson, Wiener. *Designing object-oriented software.* Prentice-Hall, 1990.

- **Se basa en un modelo de cliente-servidor**
  - **Un sistema es un conjunto de servidores que**
    - tienen responsabilidades únicas o particulares
    - proporcionan servicios a clientes
  - **Usa la idea de *contratos* que definen la naturaleza y alcance de interacciones válidas con un cliente**

  - **Contrato, colaboración**
    - para preservar encapsulación, algunos objetos deben poder hacer ciertas labores (como modificar sus valores internos) para beneficio de otros objetos
    - Ciertos servicios requieren de la colaboración de varios objetos
- **El método está basado en la responsabilidad**
  - **Durante el diseño, el enfoque es en el contrato entre el objeto cliente y el servidor**
    - El contrato especifica lo que cada objeto debe hacer
    - Y la información que comparte
  - **Contrasta con el método de diseño con objetos basado en datos**
    - que enfatizan el diseño de estructuras de datos internas a los objetos
    - y que definen herencias al considerar atributos comunes
  - **Se enfoca más a interacciones entre objetos y encapsulación**

- **Pasos para el diseño b. en responsabilidades**
  - **Enfoque incremental e iterativo.** Dos fases, seis pasos
  - **Fase exploratoria**
    - **Identifica clases**
      - * Extrae nombres y frases nominativas de las especificaciones; busca nombres que se refieran a objetos físicos, entidades conceptuales, categorías de objetos, interfaces externas
      - * Ésos son candidatos a clases
      - * También se identifican atributos de objetos y candidatos a super clases
    - **Identifica responsabilidades y las asigna a clases**
      - * ¿Cuál es el propósito de cada clase?
      - * Examina la especificación buscando oraciones de acción. Ésas son candidatos a responsabilidades
      - * Asigna responsabilidades a clases de tal forma que
        - * el trabajo esté uniformemente distribuido
        - * los métodos residan junto con la información relacionada
        - * Se compartan responsabilidades entre clases afines
    - **Encuentra colaboraciones**
      - * Examina las responsabilidades de cada clase
      - * Determina qué otras clases se requiere que colaboren para satisfacer cada responsabilidad

- **Fase constructiva**
  - **Define jerarquías**
    - * Construye jerarquías de clases — observa la herencia
    - * Las responsabilidades comunes deben flotarse tan arriba (a la superclase mayor) como sea posible
    - * Las clases abstractas no deben heredar de las concretas
    - * Construye contratos agrupando las responsabilidades usadas por los mismos clientes
  - **Define subsistemas**
    - * Dibuja una gráfica de colaboraciones para todo el sistema
    - * Busca colaboraciones frecuentes y complejas. Ellas son candidatos a subsistemas
      - * Las clases dentro de un subsistema deben soportar un conjunto pequeño y cohesivo de responsabilidades
      - * Deben ser fuertemente interdependientes
  - **Define protocolos**
    - * Detalla diseños escribiendo especificaciones (de diseño) para clases, subsistemas, y contratos
    - * Construye los protocolos (argumentos que llevan los mensajes) a los que cada clase responde

➢ **El diseño enfatiza el comportamiento dinámico y las responsabilidades de los objetos**

En vez de sus relaciones estáticas de clase

- **Contraste con el Diseño orientado a objetos (Booch)**
- **Y con los métodos de análisis orientado a objetos**
  (Especificaciones orientadas a objetos de Bailin; Análisis orientado a objetos de Coad y Yourdon; Análisis orientado a objetos de Shlaer y Mellor)
  - * Su primera fase busca construir una simulación de comportamientos e interacciones de objetos — No una jerarquía de clases

- ## Herramientas para el diseño basado en responsabilidades
  - ### Tarjeta para clase (pasos 1, 2, 3)
    Tarjeta de cartón; describe una clase. Incluye su nombre, superclases, subclases, responsabilidades, y colaboraciones
  - ### Diagrama jerárquico (paso 4)
    Un diagrama sencillo con relaciones de herencia, como un árbol o lattice. Las superclases aparecen arriba de sus subclases
  - ### Diagrama de Venn (paso 4)
    Para mostrar el traslape de responsabilidades entre clases (elipses), con el fin de identificar oportunidades para crear superclases abstractas
  - ### Gráfica de colaboraciones (pasos 4 y 5)
    Diagrama con las clases, subsistemas y contratos de un sistema, así como las trayectorias de colaboración entre ellos. Las clases son cajas; los subsistemas son cajas con esquinas redondas, rodeando varias clases. Las colaboraciones son arcos dirigidos de una clase al contrato de otra clase
  - ### Tarjeta para subsistema (paso 5)
    Tarjeta de cartón. Documenta un subsistema; incluye su nombre y la lista de sus contratos
  - ### Especificación de clase (paso 6)
    Versión expandida de la tarjeta de clase. Identifica superclases, subclases, gráficas de jerarquía, gráficas de colaboración. También incluye una descripción general de la clase, y documenta todos sus contratos y métodos
  - ### Especificación de subsistema (paso 6)
    Contiene la misma información que una especificación de clase, pero a nivel de subsistema

Componente	Diseño estructurado de Yourdon y Constantine	Ingeniería de Información de Martin	Diseño estructurado orientado a objetos de Wasserman	Diseño orientado a objetos de Booch	Diseño basado en reponsabilidades de Wirfs-Brock
Jerarquía de módulos (diseño físico)	Carta de estructuras	Diagrama de descomposición de procesos	Carta de estructuras orientada a objetos	Diagrama de módulos	*Sin soporte*
Definiciones de datos	Diagrama de jerarquías	Diagrama de modelos de datos; diagrama de estructura de datos	Carta de estructuras orientada a objetos	Diagrama de clases	Especificación de clase
Lógica de procedimientos	*Sin soporte*	Diagrama de acciones	*Sin soporte*	Esqueleto de operaciones	Especificación de clase
Secuencias de procesamiento de cabo a rabo	Diagrama de flujo de datos	Diagrama de flujo de datos; diagrama de dependencia de procesos	*Sin soporte*	Diagramas de tiempo	*Sin soporte*
Estados y transiciones entre objetos	*Sin soporte*	*Sin soporte*	*Sin soporte*	Diagrama de transiciones entre estados	*Sin soporte*
Definición de clases y herencia	*Sin soporte*	*Sin soporte*	Carta de estructuras orientada a objetos	Diagrama de clases	Diagrama jerárquico
Otras relaciones entre clases (miembro de; usa; etc.)	*Sin soporte*	*Sin soporte*	Carta de estructuras orientada a objetos	Diagrama de clases	Especificación de clase
Asignación de operaciones y servicios a clases	*Sin soporte*	*Sin soporte*	Carta de estructuras orientada a objetos	Diagrama de clases	Gráfica de colaboraciones; Epecificación de clase
Definición detallada de operaciones y servicios	*Sin soporte*	*Sin soporte*	*Sin soporte*	Esqueleto de operaciones	Especificación de clase
Conexiones de mensajes	*Sin soporte*	*Sin soporte*	Carta de estructuras orientada a objetos	Diagrama y esqueleto de objeto	Gráfica de colaboraciones

Ref.: "Object-oriented and conventional analysis and design methodologies" *Computer*, Oct 92

• Diseño por contrato

B. Meyer. *Object-oriented software construction.* Prentice-Hall. 1991

Applying "Design by Contract." *Computer,* Oct. 1992

- **Programación defensiva**
  - **Incluye pruebas y verificaciones contra todo lo que pueda estar erróneo — argumentos erróneos**

    Problemas: 1) El que llama y el llamado checan — doble chequeo. 2) Tanto código obscurece el cálculo, y es fuente de nuevos errores.
  - **Solución: Un contrato entre el que llama (cliente) y el llamado (proveedor o contratista)**

    Se especifican las obligaciones de cada quien.
- **Contrato**
  - **Especifica qué cosas debe hacerse: el cliente espera cierto resultado**
  - **Especifica qué tan poco es aceptable: el contratista no se obliga a más de esto**
  - **No hay "cláusulas escondidas": si el cliente satisface sus obligaciones, el contratista *debe* hacer lo convenido.**

Sujeto	Obligaciones	Beneficios
Cliente	Proveer una carta o paquete de no más de 5 Kg y no más de 2m en cualquier dimensión. Pagar 100 francos	Su paquete será entregado al destinatario en París en cuatro horas o menos
Proveedor	Entregar paquete al destinatario en París en 4 horas o menos	No necesita lidiar con paquetes demasiado grandes, pesados o sin paga

- **Afirmaciones**
  - **Pre-condiciones**
  - **Post-condiciones**

```
nombre-de-rutina (argumentos) is
— Comentario encabezador
require
        pre-condición (predicado)
do
        Cuerpo de la rutina (instrucciones)
ensure
        post-condición (predicado)
end
```

  - **Ejemplo:**

```
put_child (nuevo: NODE) is
— Agrega nuevo a los hijos del nodo actual
require
        nuevo <> Void
do
        ... algoritmo de inserción
ensure
        nuevo.parent = Current;
        child_count = old child_count + 1
end    put_child
```

- El papel de las afirmaciones
  - No describen casos especiales o raros pero válidos
  - Describen casos inválidos: errores

    Para manejar casos especiales o raros pero válidos, inclúyanse en un IF dentro del cuerpo de la rutina. No se incluyan en las afirmaciones

    * Una condición está en el **require** o en el cuerpo, nunca en ambos. — *Esto es exactamente opuesto a la programación defensiva.*

    * Pero trabaja. No hay necesidad de verificaciones redundantes.
  - Cualquier violación durante la ejecución de una afirmación es un error en el software

    * Una violación de una pre-condición es un error del cliente (o llamador). El cliente no observó las condiciones impuestas a las llamadas correctas.

    * Una violación de una post-condición es un error del proveedor (la subrutina). La subrutina no cumplió con lo prometido.

- ¿Quién debe checar?
  - El cliente (si la afirmación es una pre-condición) o el proveedor (subrutina) (si la afirmación es una post-condición). NO AMBOS
  - Es una decisión pragmática
    * Si ambos checaran, todo sería muy lento
    * Si la pre-condición es muy fuerte, más trabajo para el cliente
    * El lenguaje Eiffel se inclina por chequeo fuerte por el cliente. Cada rutina se concentra en hacer bien algo bien definido: "su contrato". El cliente no espera milagros.
  - Posibles objeciones: cada cliente debe hacer chequeos

    a) Esto no es cierto. Cada cliente debe *tan solo* asegurar cierta condición. A lo mejor ya estaba checada antes.

    b) Si cada cliente en realidad debe hacer chequeos, entonces

    b1) Todos checan lo mismo. Se trata de un diseño pobre de la interfaz de la subrutina. Hay que renegociar el contrato y hacer más tolerante la pre-condición, a fin de incluir en el cuerpo de la rutina lo que todos los clientes checaban.

    b2) No todos checan lo mismo. Entonces el diseño está bien — cada cliente verifica algo distinto que la rutina no necesariamente debe o puede verificar.

    * Hay casos anormales que una rutina no puede manejar, pero el cliente sí. Ejemplo: ¿Qué hacer cuando me piden entregar el elemento a la cabeza de la pila, pero ésta se encuentra vacía? No hay regla general. El cliente debe manejar este caso especial — no la subrutina.

Invariantes de clase

- **Son afirmaciones que se aplican a todos los elementos de una clase.**

```
class BINARY-TREE [T] feature
    ... declaraciones de atributos y rutinas ...
invariant
    left <> Void implies (left.parent = Current);
    right <> Void implies (right.parent = Current)
end — class BINARY-TREE
```

- **El invariante debe satisfacerse después de la creación de cada miembro de la clase**
- **El invariante debe preservarse por cada método público de la clase (esto es, por cada subrutina disponible a clientes)**

  Cada subrutina debe satisfacer el invariante a su salida si éste fue satisfecho a la entrada

  * De hecho, cada invariante es agregado como un AND de cada pre-condición y post-condición de cada método público

- **Vigilando las afirmaciones**

  ¿Qué sucede si, durante la ejecución, se viola una afirmación?

  - **Depende de la opción con que se compiló:**
    * No verifiques afirmaciones
    * Verifica precondiciones solamente (se escoge por omisión)
    * Verifica precondiciones y postcondiciones
    * Verifica precondiciones, postcondiciones e invariantes de clase
    * Verifica todas las afirmaciones (todo lo anterior más otras afirmaciones, como invariantes de ciclos — no se han explicado)
  - **Su efecto es señalar una excepción**

    Se verá más adelante qué es y para qué sirve una excepción

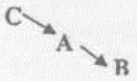
- **¿Para qué vigilar?**
  - **Para encontrar errores en la programación**
  - **Para guiar dónde están, y poder corregirlos**

- Efecto de las afirmaciones sobre la herencia
  - Redeclaración
    - Se redefine un método o variable ya existente
    - Se hace efectivo un método abstracto que no se había implementado ("effecting")
  - Polimorfismo
    - La herencia controla cuál de varios métodos con el mismo nombre escoger. Requiere de una liga dinámica.
  - Liga dinámica

En todos estos casos, una redeclaración no debe contradecir (ser incompatible con) la semántica de la declaración original

- Los contratos ayudan a verificar que la redeclaración conserve la semántica original
- Una redeclaración es un subcontrato — Un cliente llama a A. A subcontrata su trabajo a B.

  El subcontratado (B) no debe engañar al cliente, haciendo algo que no se esperaba de (A). El cliente no sabe que (A) no lo hizo. El cliente espera que (A) — en este caso, (B)— cumpla con el contrato de (A).



- No se deben permitir:
  - Que B suponga precondiciones más fuertes — el cliente no las esperaba
  - Que B satisfaga postcondiciones más débiles — el cliente puede recibir resultados insatisfactorios, ilegales desde el punto de vista del contrato de (A).

- Sí se pueden permitir
  - Que B tenga una precondición más débil — menos severa — que A
  - Que B tenga una postcondición más severa que A

Ambos casos resultan en un *mejor* trabajo hecho por B del que se esperaba de A

- Cómo manejar situaciones anormales (excepciones)
  - No es aceptable simplemente imprimir "se ha cometido un error" y regresar un resultado inválido (que no cumple la postcondición). Hay que hacer uno de estos tres:
    - Intentar de nuevo: Se deben poner los objetos en un estado estable [que cumple con los invariantes] e intentar nuevamente. (El contratista perdió la batalla)
    - (El contratista perdió la guerra). Poner los objetos en un estado estable y reportar una falla al cliente
    - (Falsa alarma). Tomar alguna acción correctiva e intentar de nuevo

■ Ejemplo

```
pide_entero_al_usuario: INTEGER is
    — lee un entero (permite al usuario hasta cinco intentos)
local
    fallas: INTEGER
do
    Resultado := getint
rescue
    fallas := fallas + 1;
    if fallas < 5 then
        message ("La entrada debe ser un entero. Favor de inten
                 tar de nuevo");
        retry
    end
end — pide_entero_al_usuario
```

- **Método de módulos rígidos con acoplamientos flexibles**

  Adolfo Guzmán, SoftwarePro International. 1991.

  - **El método es ortogonal a los otros. Puede usarse junto con cualquiera de los otros**

    Hace buena unión con métodos que usan objetos (por ejemplo, el de diseño basado en responsabilidades)

  - **Preferiblemente antes de usar otro (ver más adelante)**

- **Objetivos:**

  - **Manejar el riesgo. Al identificar**
    - **Requerimientos o especificaciones dudosas o probablemente cambiantes**
      - El cliente no está seguro de algunas partes de lo que quiere. O nosotros no estamos seguros de haber entendido esas partes
      - Probablemente haya cambios en algunas especificaciones
    - **Partes de la solución o diseño dudosas o inciertas**

      Nosotros (los diseñadores) no confiamos en determinado componente de una posible solución. Tal vez no sirva.

- **Consta de dos fases, que pueden desarrollarse parcialmente en paralelo**
  - **Clasificación de especificaciones en estables y tentativas**
  - **Identificación y resolución de componentes críticos**

- **Fase: Clasificación de especificaciones en estables y tentativas**
  - **Paso 1: Revisar cada especificación**

    Revisar cada especificación (obtenidas en la etapa de análisis) separándola en
    - **Estable o definitiva**

      Aquéllas que es probable o seguro que no cambie
      * Porque el cliente está satisfecho con ella
      * Por experiencias anteriores (nuestras o del cliente)
      * Por otras razones
    - **Tentativa o incierta (y estimar el tamaño de la incertidumbre o rango de cambio probable)**

      Aquélla que es probable o seguro que **sí** cambie
      * Porque en la etapa de análisis se analizó superficialmente
      * Porque sea difícil de implementar o satisfacer
      * Porque el cliente no la entienda
      * Porque nosotros no le entendamos
      * Por ser incompatible con otra especificación más importante
      * Por otras razones
  - **Paso 2: Elaborar un diseño tomando en cuenta la clasificación del paso 1**

    Diseñar la arquitectura del sistema como un conjunto de módulos rígidos (no particularmente fáciles de cambiar) unidos por módulos o acoplamientos flexibles (diseñados para ser fáciles de cambiar)
    - **Los módulos flexibles tratan de satisfacer, mediante su cambio, a las especificaciones tentativas**
    - **Se prevé que con ciertos cambios en los módulos flexibles se hará frente a cualquier cambio de una especificación tentativa**

- **Módulos flexibles**
  - **No es necesario que cada módulo flexible corresponda a una especificación tentativa**
  - **A veces la flexibilidad consiste en la manera de interconectar los módulos rígidos**

    Es como si el programa principal o "driver" fuera el único módulo flexible

Úsense algunos de los siguientes recursos:

- **Archivo de inicialización o definición**

  Permite ajustar a la medida el comportamiento del programa, trayectorias de archivos ("pathnames"), cambio de valores por omisión, color del fondo, etc.

  Útil para cambios ligeros y medianos
- **Fórmulas introducidas por el usuario**

  Para cambiar la definición de ciertas operaciones. Ejemplo: Sumarizador de Conalep.

  Requiere de un intérprete o de un compilador dinámico
- **Usar el diccionario de datos para tomar información fresca o cambiante**

  Por ejemplo, nuevas tablas de la base de datos. Ejemplo: Consultor amigable con preguntas espontáneas, Conalep.
- **Descriptor de archivos**

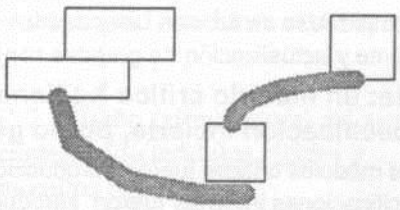
  Ejemplo: RepCob, CFE.
- **Programa para definición dinámica (por el usuario) de nuevos formatos a captar**

  Ejemplo: Sumarizador, Conalep
- **Gramática para definir algún módulo flexible**

  Uso de Yacc y Lex

■ Módulos rígidos

- Se programan convencionalmente — con pocas alternativas para cambios
- Tener cuidado de su interacción con los módulos flexibles
  - Deben poder usar o accesar los módulos flexibles en forma "general" (es decir, aunque el módulo flexible haya cambiado)
  - No vayamos a introducir rigidez por la manera particular en que **hoy** estamos usando o llamando a algún módulo flexible



Se usan componentes rígidos para resolver partes que se consideran estables

Los componentes flexibles aportan resistencia y adaptación al cambio

- Fase: Identificación de componentes críticos
  - Paso I: Analizar el diseño (arquitectura) obtenido en la fase anterior, identificando cada módulo o componente cuyo desempeño sea crítico porque
    - Se duda si van a poder hacer lo que de él se espera
    - No se le conoce; no se tiene experiencia en su uso
    - Se desconoce si se va a poder interconectar bien
    - Por alguna otra razón se teme que no funcione a satisfacción

      Ejemplo: Uso de Dbase para un sistema de contabilidad

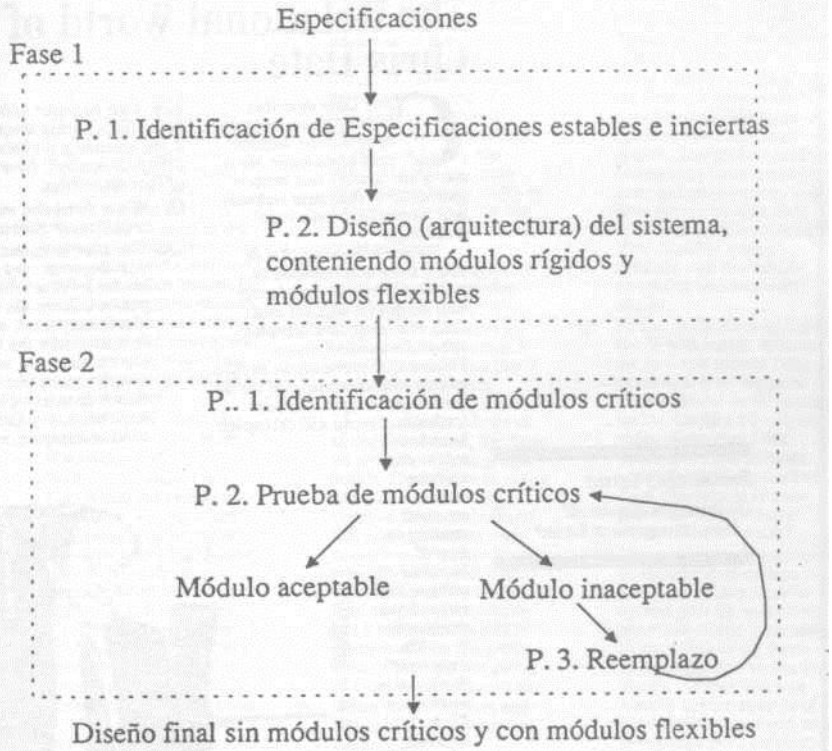
      Ejemplo: Uso de árboles balanceados ("B-trees") para almacenamiento y actualización de grandes conjuntos de información
    - Nota: un módulo crítico no tiene que ver con una especificación incierta, por lo general

      Los módulos críticos fueron introducidos por el diseñador. Las especificaciones inciertas fueron introducidas por el cliente o usuario.
  - Paso 2: Para cada módulo crítico, hacer pruebas que permitan definirlo como bueno o aceptable, o bien descartarlo
    - Escribir un programa de prueba real (cantidades de registros, complejidad; manejo de varios archivos)
    - Medir cuantitativamente (no mediante opiniones o estimaciones) si cumple con lo requerido

      * Hacer cambios en sus parámetros, buscando mejor desempeño
    - Nuestro diseño no debe tener muchos módulos críticos, ya que nunca acabaríamos de probarlos

- Paso 3: Reemplazar los módulos críticos no aceptables (encontrados en el paso 2) por otros, y, si es necesario, repetir el Paso I o cambiar el diseño (arquitectura) obtenido de la fase anterior

- **Diagrama del método de módulos rígidos y flexibles**



Interface

!Candle

Candle's view on IBM's database world

Special Supplement to
Database Programming & Design

The Relational World of Chris Date

Chris Date describes himself as an independent author, lecturer, and consultant. He is one of the world's best known specialists in database technology, particularly relational technology.

Prior to leaving IBM in 1983, Date was involved in technical planning and externals design for SQL/DS and DB2. His book, "An Introduction to Database Systems: Volume I" is currently in its fifth edition and is considered the de facto standard for database textbooks. Nearly 450,00 copies have been sold to date and the book is required reading in several hundred colleges and universities worldwide.

We were most fortunate to track down Chris Date between speaking engagements and asked him, tape recorder in hand, if he wouldn't mind answering a few questions on behalf of Interface readers. Here is some of that discussion.

Q: What does the word 'relational' mean to you?

CJD: The answer to that question depends very much on the level at which you're talking. On the elementary level, when I first introduce the idea in my classes, I say 'relational' simply means that all the data is seen as tables, which is a very familiar kind of structure, and then

"If you're a database professional, and you don't know the relational model thoroughly, you are practicing medicine without a license."
– Chris Date

Inside this issue

For further information, circle 10 on the reader service card.

you have operators that operate on those tables, and the operators are essentially cut-and-paste operators for tables. So you can paste tables together with a join and make another table, or you can cut tables up, with restrict and project operations, and again make smaller tables.

If you take relational to the next level of depth, I would say relational means you have to support everything in the relational model, which of course is much more than just tables, restrict, project and join. When I say all parts of the relational model, obviously we could spend a day on saying what that means. In fact, I have written several papers that explain what I think that level means, and I don't think it would be appropriate to go into all that detail here.

Q: How did you "sell" the popularity of the relational model?

CJD: Well, as a matter of fact, I think that the relational advocates, among which I certainly include myself, did ourselves a big disservice in the early days, because when the ideas of relational first came along, we had to convince the community that this was a good thing to do and, in particular, had to talk about how relational could do all the things that the pre-existing database systems did. So we got into the business of comparing relational with the hierarchic and network "models" found in IMS and IDMS and so on.

As a result of such comparisons, people got the idea that relational was the same kind of animal as hierarchic and Network. The impression was that we were talking about the same sort of thing, just another step into the future. But relational really isn't the same kind of animal at all. Rather, I regard relational database as the foundation, the principles that any database system really ought to adhere to. Even if, on the outside, the system looks quite different.

Q: How would the system look different on the outside?

CJD: Suppose we consider an object-oriented system. I would maintain that the inside of that system should still be relational. To repeat, I see relational as the foundation, the basis on which everything else builds. The trouble is, many (database technology) vendors don't understand this point, let alone their customers. Vendors have typically marketed "object oriented" as if it's a new way of solving problems, but it really isn't, and I'm concerned that the problems of the past are going to come back to haunt us.

Q: What flaws do you see behind object-oriented databases?

CJD: An analogy I draw is the following: A database system that doesn't support the operators of relational algebra, doesn't support them fully or doesn't support them correctly, is just like a computer that doesn't support arithmetic properly. We all do wonderful things with computers, but inside it simply performs basic arithmetic. Those arithmetic functions have to be there.

In the same way, inside the database system we must have the relational operators and relational objects. And to the extent the vendors have failed to meet this requirement, we have problems with their systems, which lead us to some horrible workarounds, and sometimes surprising and incorrect results.

Q: In the early days of DB2, there was much debate on to what extent DB2 conformed to the relational model as defined by Dr. Codd's 12 original rules. Now that DB2 dominates the RDBMS area of the MVS market, how important is conformance with those theoretical models?

CJD: There are several questions all mixed together here, and what I'd like to do is try and pick the questions apart and tackle them one by one. First of all, those 12 rules did not "define the model." All they did was identify certain aspects of database management, not necessarily even relational aspects, but just aspects of database management that Ted Codd happened to think were important back

For further information, circle 10 on the reader service card.

in 1985. And as we know, the publication of those 12 rules led to a huge amount of brouhaha in the marketplace, and I think in some cases some rather unseemly behavior.

I felt at the time, and still feel, that those 12 rules should have been regarded just as a kind of first cut, a sort of trial balloon, if you like, for all kinds of reasons; I'd like to run through some of the reasons, quickly.

First of all, it was never really clear why just those 12 things were important. There are numerous other things you could think of that seemed to be equally important that weren't in the list – for example, dynamic data definition. Which was something that was new with relational, I might mention, although we're used to it now.

Another thing was, those 12 rules were not all independent of each other. Ted never said they were; but there certainly was a general idea that they ought to be. If they're not independent, for example, if say rule eight depends on rule four and you score a no on rule four, you're necessarily going to score a no on rule eight. In other words, it's not really a fair evaluation of a system.

Ted's idea behind publishing those rules was to lay out a list of items that (a), were technically achievable in the first place, and (b), would provide clear practical benefits to the users if they were in fact achieved. But it turned out that some of them were not technically achievable after all. And some of them were not really worded carefully enough, which meant that one couldn't really tell whether a system satisfied the rule or not. And in at least one case the rule was at much too coarse a level.

Q: An example?

CJD: There was a rule that said you must be able to update views. I think it would have been much better to break that down and say, you must be able to update restriction views, you must be able to update projection views, you must be able to update join views, and so on; because some products can update some views and not others.

Moreover, the wording "you must be able to update updatable views" very clearly suggests that we know precisely which views are updatable; but the fact is we don't. It's still partly a research problem to solve that one.

Also there was at least one rule that was worded in terms of the future: If and when the system supports distributed databases, then certain capabilities have to be provided. Every system could get a 'yes' on that one, because it was talking about future objectives.

Q: And along with those rules, there was a rating scheme...

CJD: Ah yes. We got into these silly things where vendor A would say, ah, we're now 46% relational, and vendor B would say we're 48% relational and so on. It was not a bad idea to have a checklist of functions for evaluating systems. Customers and vendors can use such a list to see if a system can perform function X, and so on. But I don't think there is an objective measure that would say what the definitive value of individual items is. It's different for different people. In other words, the rating scheme was misguided.

Q: Is it still important to conform to a theoretical model?

CJD: Absolutely. It's very important as long as you're talking about the *right* theoretical model. I believe we must conform to the relational model because the relational model is the foundation.

At the risk of being severely misunderstood, I sometimes say that the relational model is like the assembly language of database systems. It's not the ultimate goal, it's the beginning, not the end. It's the base on which to build. The idea is if the system supports the fundamentals of the relational model, in other words, conforms to the model properly, then you can build higher level interfaces on top, that are really useful.

Finally, your question concludes, "now that DB2 dominates the MVS market." While that is true, the

question suggests that DB2 is the de facto relational standard. It seems to me that precisely because IBM has the dominant position, it has a huge social responsibility to do the right thing! In the context of our discussion, that means conform to the relational model!

Q: Is the relational model the end of the line in DBMS design?

CJD: Definitely not! Like I said before, it's the beginning, not the end. Relational was never intended to be seen as the ultimate goal. Though it's ironic that we don't have any products which have fully achieved that "beginning" goal yet.

Q: How will DBMS be affected by development in the world of object-oriented programming and design?

CJD: Object-oriented ideas clearly will affect relational database management systems. Right now we have two technologies out in the marketplace, relational and object oriented. Both have some good ideas and we should avoid thinking of them as being in conflict. I dislike the talk about battles and wars in the trade press. Instead, we need to be looking for ways to marry the two technologies together. And indeed there are several vendors working on exactly that idea.

Mind you, although object oriented has some very good ideas, it also has some very bad ideas. And I think relational systems should be extended to include the good ideas, but certainly not at the expense of incorporating the bad ones as well.

Q: The good ideas?

CJD: I think the good ideas are: user-defined data types, user-defined functions, and inheritance.

Q: The bad ideas?

CJD: I think the idea of encapsulation is bad, at least if taken to extremes. Encapsulation is regarded as a virtue in object-oriented systems, but I think it's possible to go overboard with the encapsulation idea. If everything is encapsulated, which implies that the only way you can operate on data is through pre-defined methods or functions, one consequence is that ad hoc query is impossible. You can't do anything that hasn't been thought about ahead of time. Now that is clearly extreme. You have to be able to do ad hoc query as well, which means breaking the encapsulation idea.

Second, I don't believe in object-IDs because they fail to do away with the need for user keys (or primary keys, if you prefer). I certainly don't want object IDs visible in my relational interface, because an object ID is conceptually nothing but a pointer, and you find yourself writing pointer-chasing code just like we did in the old days of CODASYL.

That leads to another concern – the highly programmer-oriented nature of OO systems. OO seems to be very much a programmer's perception of what database is about. This seems a giant step backwards. All the business of optimization which we've worked so hard on in relational systems is effectively thrown away, because now performance is largely back in the hands of the application programmer.

I also don't like the idea of bundling functions with objects. I see no need for that, I think it's clumsy, it's asymmetric, it's artificial, and it's awkward. And I certainly don't like the fact that integrity support has to be done procedurally.

Q: Of the good things you mention in object-oriented technology, aren't they valuable?

CJD: Yes but my problem is, those good ideas are not new. We were talking about exactly those things in the relational world at least 15 years ago. I do find it a little frustrating to see how our industry is so much driven by fluff not substance. Suddenly, somebody labels an existing idea object-oriented, and everybody thinks they want an object-oriented system. Everybody loves object-oriented but not many people really know what it is.

If the relational vendors had implemented all the ideas in the relational

For further information, circle 10 on the reader service card.

Good ideas { • user-defined data-types • " " functions • inheritance

bad { • encapsul • object-ID • [programm w objec

• bundling funct w objects

model and if they'd done it correctly, I don't think we'd be talking about object-oriented systems today. Relational systems are perfectly capable, in principle, of addressing the kinds of problems that object-oriented systems claim to address. That's not to say that today's relational products can do it, they cannot.

What I'm really objecting to here is an argument that goes something like this: first, relational's no good, because it doesn't support some feature, X, let's say, where X actually is included in the theory but not in the products; so let's go and invent something new, let's say object-oriented that does support X; and now we have something that is "better than relational." It's not only not better than relational, it's probably worse, because it very likely does not include all the good things that relational does have, like closure, for example, which is a major missing feature in object-oriented systems.

Q: How do you think relational systems can be extended to take advantage of OO's ideas?

CJD: The fundamental construct in object oriented is the object class. An object class conceptually is a user-defined data type, of arbitrary complexity – and of course I'm using the term data type in the modern sense, which means it includes operators. The system knows what operators apply to objects of that particular type – but basically an object class is a user-defined data type of arbitrary complexity. Now if I switch over to the relational world, one of the big things in the relational model is the domain. Domains typically have not been implemented in most relational products, but nevertheless domains are a very fundamental ingredient of the relational model. And a domain is a user-defined data type of arbitrary complexity. In other words, it's the same thing!

So a domain in the relational world is the same as an object-class in the object-oriented world. Which means, the key to marrying these two technologies is domains. A relational system that implemented domains right would be able to address all the problems that the object-oriented folks say relational systems cannot address. So I conclude that domains are the key to the marriage.

However, if you look at the various products and prototypes that are trying to marry these technologies together, you will find they are making what I regard as a fundamental mistake. They are not equating domains and object-classes, they're equating relations and object-classes. They say that a relation is an object-class and each row in relation is an object-instance of that class.

Although that sort of mapping or parallelism looks superficially quite attractive and simple, it breaks down as soon as you do a join, for example. Suppose I have a department object-class, which we think of as a department relation, and an employee object class, which we think of as an employee relation. We join them together and what object class is the result? That question is unanswerable.

I've attended several presentations by object-oriented people. Typically, on the very last slide, they list "unsolved problems" They tend to dismiss "what object class is the result of a join?" as if this is a small problem that can be solved later. To find a flaw of such magnitude at the very foundation of these products does not bode well for the future.

Q: Is anyone listening to your concerns?

CJD: I learned only last week something interesting about SQL 3, the proposed next step for the SQL standard but still several years in the future. One of the things they're proposing to do is to add object-oriented extensions to SQL. For quite a long time, they were proposing this mapping that I feel is wrong, equating tables or relations and object-classes. But very recently, they switched around and agreed that it was wrong.

Q: In light of your concerns about flaws in new

technologies and hasty moves to them, how can DBAs and database planners be forearmed and forewarned?

CJD: If I knew the perfect answer to that question I'd be a very rich man. The best answer is clearly education. If you're a database professional, and you don't know the relational model thoroughly, you are practicing medicine without a license. You must know the principles of your field. I've been working on this stuff for over 20 years now, and I'm still discovering new things. If you can get to be sufficiently knowledgeable you might have some influence on the way the field develops. Try and find a friendly member of the SQL standards committee who you can help you get some input to the committee. Learn and refresh yourself on database design principles, application design principles, and product specifics. Only then can you start exercising influence and lean on your vendor. For example, if you work in a DB2 shop, lean on IBM to implement the things they haven't done yet, like domains, for example, integrity constraints or business rules, and stored procedures.

Q: Your views on deficiencies in DB2 SQL are widely published. Version 2.3 introduced the OPTIMIZE FOR n ROWS clause, which allows developers to try to influence the optimizer directly in its choice of access path. Rather than having to rely on surreptitious techniques, such as massaging catalogue statistics, does this imply a weakness in the DB2 optimizer or do the demands of high-performance transaction systems require more procedural intervention by developers?

CJD: There will always be cases where the user knows more than the optimizer does. I don't see any way around that. Therefore, it follows that having a way for the user to help the optimizer or give the optimizer a hint is okay in principle, but there are some things you have to watch for. First of all, of course, it mustn't violate relational principles. In particular, it mustn't violate data independence. We don't want a clause that says, 'use index X for this query.' That would be very bad, because now the query won't run if index X isn't there. Even if index X is there and it's not the best way to do the query, it's not a good idea.

Second, we don't want relational database vendors to use this approach as an excuse not to do their best to make the optimizer as good as possible. And third, you are now making yourself dependent on the vendor – at least on that particular system.

Turning to the specific case you mentioned, OPTIMIZE FOR n ROWs, I don't have any problem with that apart from what I've just said. Logically it's a no-op. It doesn't make any difference to the logic of your program whether the clause is there or not. In fact, it occurs to me that that's another general principle. Any such hints to the optimizer must be logically a no-op, it mustn't actually change the logic. Your very last clause in that long question talks about procedural intervention; if that means what I think it means, I object! We don't want to do anything procedurally if we can possibly help it; we want to do it declaratively if we can. And OPTIMIZE FOR n ROWs is at least declarative; you're not writing procedural code to make the optimizer do something. Declarative is always better than procedural, if it's feasible.

I do understand that there are some cases where it's not feasible. An example might be bill-of-materials processing. You can't do that declaratively in today's relational products, because the necessary operators are not there. So you have to write loops or use other techniques, such as recursion. But even that problem (bill of materials), is not inherently procedural. There are declarative solutions to it, and it is a well understood problem in the research world. It's a solved problem, it is just that the funcionality has not reached the vendor products yet, with one or two exceptions.

For further information, circle 10 on the reader service card.