PROGRAMACIÓN E INGENIERÍA DE SOFTWARE

A. ALGORÍTMICA

I. Representación y manipulación de datos

1. Taxonomía de las estructuras de datos

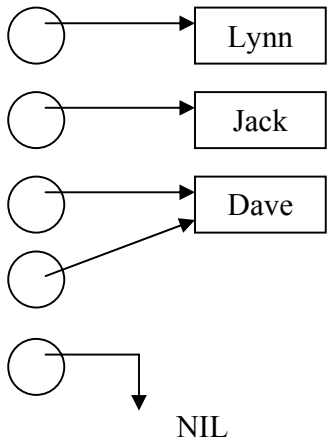
- ***Apuntadores***

Un apuntador (también llamado liga) se de fine como una variable que indica la localización de alguna otra variable; por ejemplo: alguna sección de datos. Así, un apuntador de pila indica la localización del elemento en el tope superior de la pila y los apuntadores de cola dan los sitios de la cabeza y el extremo de la cola.

Estructuras dinámicas de datos.

Este tipo de estructura es generado a partir de un tipo de dato conocido con el nombre de puntero o de referencia. Un dato puntero almacena una dirección o referencia a un dato. Por tanto, debe distinguirse entre un dato puntero y el dato al cual se apunta. Se usara la notación P=^D para indicar que P es un puntero a datos de tipo D. Se genera un valor para una variable de tipo puntero cuando se asigna dinámicamente, memoria a un dato apuntado por ella. La principal ventaja de los punteros o apuntadores es que se puede adquirir posiciones de memoria que se necesitan, y liberarlas cuando ya no se requieran. De esta manera se pueden crear estructuras dinámicas que se expandan ose contraigan, según se les agregue o elimine elementos. El dinamismo de estas estructuras soluciona el problema de decidir cual es la cantidad optima de memoria que debe reservarse para un problema dado. Al proceso que asigna y reasigna la memoria en esta forma se le conoce como asignación dinámica de memoria.

Estructura con apuntadores Nil



Dos apuntadores pueden referirse al mismo registro o un apuntador no puede referirse a ningún registro.

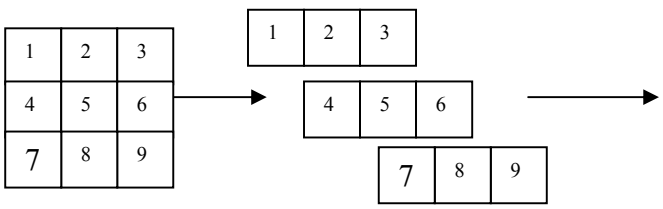
- ***Vectores, matrices, cubos, hipercubos***

Vectores o Arreglos

Un arreglo o vector contiene un numero constante de posiciones, es decir su tamaño se fija cuando el programa se compila. Arreglo o vector el cual contiene un numero constante de posiciones. Los arreglos también se corresponden directamente con los vectores, termino matemático utilizado para las listas indexadas de objetos. Análogamente , los arreglos bidimensionales se corresponden con las matrices.

Arreglos Rectangulares

Casi todos los lenguajes de alto nivel proporcionan medios convenientes y eficaces para almacenarlos y acceder a ellos. El almacenamiento en la computadora se arregla en una secuencia contigua, es decir, en una línea recta y cada entrada sigue a la otra. La forma de leer un arreglo rectangular, es leer las entradas del primer renglón de izquierda a derecha, después las entradas del segundo renglón así hasta terminar de leer el ultimo renglón.

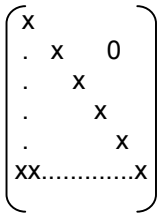


Este es el orden en que la mayoría de los compiladores almacenan un arreglo rectangular, y se denomina ordenamiento por renglón mayor.

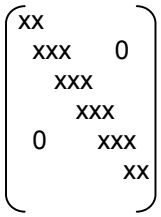
Matrices

Arreglos Triangulares

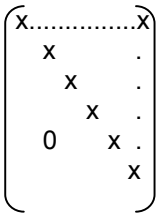
Una matriz triangular inferior puede definirse formalmente como un arreglo cuadrado en el cual la entrada es 0 en cada posición donde el índice de columna sea mas grande que el índice del renglón.



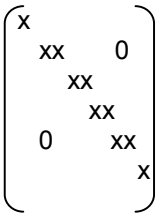
Matriz Triangular Inferior



Matriz Tridiagonal



Matriz Triangular Superior

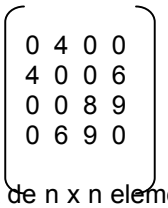


Matriz Diagonal

La diagonal principal de una matriz cuadrada se compone de las entradas para las cuales los índices de renglón y de columna son iguales. Una matriz Diagonal es una matriz cuadrada en la que todas las entradas que no están sobre la diagonal principal son ceros. Una matriz tridiagonal es una matriz cuadrada en la que todas las entradas son cero excepto posiblemente aquellas sobre la diagonal principal y sobre las diagonales inmediatamente arriba y debajo de ella. Es decir, T es una matriz tridiagonal si T [ij] = 0 a menos que | i – j| <= 1. La transpuesta de una matriz es la matriz obtenida al intercambiar sus renglones por columnas correspondientes. Es decir, que la matriz B sea la transpuesta de la matriz A significa que B[j,i] = A[i,j] para todos los índices (permitidos) i y j. Una matriz triangular superior es aquella en que todas las entradas bajo la diagonal principal son cero. La transpuesta de una matriz triangular inferior será una matriz triangular superior.

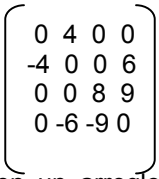
Matrices Simétricas y Antisimétricas

Una matriz A de n x n elementos es simétrica si A [i,j] es igual A[j,i], y esto se cumple para todo i y para todo j.



Matriz simétrica

una matriz A de n x n elementos es antisimétrica si $\overline{A}$ [i,j] es igual a –A [j,i], y esto se cumple para todo i y para todo j; considerando a i <> j.



Matriz antisimétrica

para almacenar en un arreglo unidimensional una matriz simétrica, se puede hacer almacenando solamente los elementos de una matriz triangular inferior o superior; por ejemplo: almacenando la matriz triangular inferior .

A [1,1] A [2,1] A [2,2] A [3,1] A [3,2] A [3,3] A [4,1] A [4,2] A [4,3] A [4,4]

0	4	0	0	0	8	0	6	9	0

i >= j

1 2 3 4 5 6 7 8 9 10

por otra parte, para almacenar una matriz antisimétrica, se procede de la misma forma almacenando solamente los elementos de la matriz triangular inferior o superior.

A [1,1] A [1,2] A [1,3] A [1,4] A [2,2] A [2,3] A [2,4] A [3,3] A [3,4] A [4,4]

0	4	0	0	0	0	6	8	9	0

i >= j

1 2 3 4 5 6 7 8 9 10

- ***Listas: simples, ligadas, doblemente ligadas, colas, pilas (stacks), colas de prioridades***

Listas

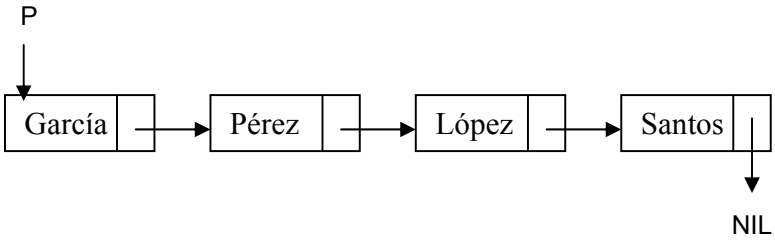
Una lista es una colección de elementos llamados generalmente nodos. El orden entre los nodos se establece por medio de punteros, es decir, direcciones o referencias a otros nodos. Un nodo consta de dos partes:

- Un campo información que será del tipo de datos que se quiera almacenar en una lista.
- Un campo liga de tipo puntero que se utiliza para establecer la liga o el en lace con otro nodo de la lista.

Si el nodo fuera el ultimo de la lista este campo tendrá como valor: nil (vacío).

Al emplearse el campo liga para relacionar dos nodos no es necesario almacenar físicamente a los nodos en espacios contiguos.

Ejemplo de lista

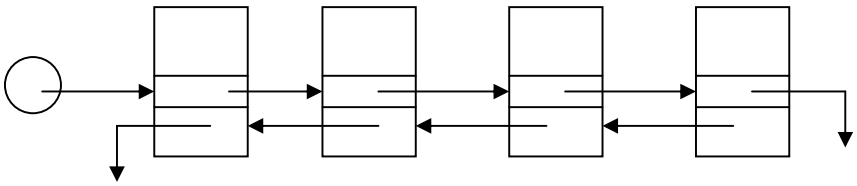


El primer nodo de la lista esta apuntado por una variable P, de tipo puntero (P almacena la dirección del primer nodo). El campo liga del ultimo nodo de la lista tiene un valor nil que indica que dicho nodo no apunta a ningún otro. Las operaciones que pueden llevarse acabo en una lista son:

- Recorrido de la lista.
- Inserción de un elemento.
- Borrado de un elemento.
- Búsqueda de un elemento.

Lista doblemente ligada

Una lista doblemente ligada es aquella en que cada nodo tiene dos ligas, una para el siguiente nodo de la lista y otra para el nodo precedente. De este modo es posible moverse en cualquier dirección atraves de la lista conservando solo un apuntador.



Lista doblemente ligada con apuntadores.

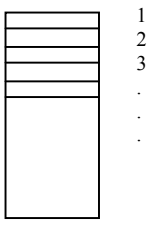
Colas

La cola se define como una lista en la que todas las adiciones a la lista se hacen a un extremo y todas las eliminaciones se efectúan en el otro extremo. Las colas se llaman también primero en entrar, primero en salir (first-in-first-out) o PEPS (FIFO). Al elemento en una cola lista para manejarse, es decir, el primer elemento que se removerá de la cola, lo llamamos cabeza de cola o frente de la cola, y el ultimo elemento, es decir al más reciente agregado, extremo o fondo de la cola. Las operaciones de una cola son:

- Determinar si la cola esta vacía o no.
- Recobrar el primer nodo de la cola, si no esta vacía.
- Insertar un nuevo nodo después del ultimo nodo de la cola.
- Eliminar el primer nodo de la cola, si no esta vacía.

Pilas

La pila se define como una lista en la que todas las inserciones y eliminaciones se hacen por un extremo, llamado tope de la pila. Cuando agregamos un elemento a la pila decimos que se le inserta “empujándolo” en la pila y cuando sacamos un elemento decimos que se le extrae de la pila. Otro nombre que se usa para en ocasiones para designar pilas es el de listas UEPS (LIFO),basado en la propiedad último en entrar primero en salir (last in, first out).



Tope
Maxpila es una constante que da el máximo tamaño permitido de la pila.

```
Type pila = record
        Tope: 0 ....maxpila;
        Entrada: array [ 1 . . .maxpila] of elemento
End;
```

Cuando queremos insertar un elemento en una pila llena ocurre un desbordamiento y cuando queremos extraer un elemento de una pila vacía es insuficiencia. Las operaciones de una pila son:

- Determinar si la pila esta vacía o no.
- Recobrar el ultimo nodo de la pila, si no esta vacía.
- Insertar un nuevo nodo después del ultimo nodo de la pila.
- Eliminar el ultimo nodo de la pila, si no esta vacía.

Colas de prioridad

Una cola de prioridad es una estructura de datos con solo dos operaciones:

1. insertar un elemento.
2. suprimir el elemento que tenga la llave más grande (o la más pequeña).

Si los elementos tienen llaves iguales, la regla habitual establece que el primer elemento introducido deberá ser el primero en eliminarse. Por ejemplo: en un sistema de computo de tiempo compartido, un extenso numero de tareas puede estar esperando para hacer uso de la CPU. Una de ellas tendrá mayor prioridad que otras. De ahí que el conjunto de las que esperan formen una cola de prioridad.

- ***Árboles: binarios, n-arios***

Árboles

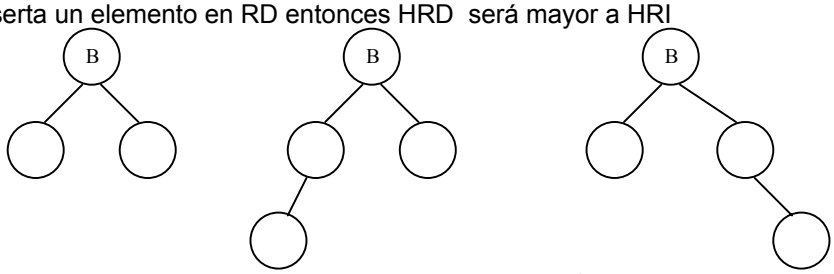
Árboles Balanceados

Se define un árbol balanceado como un árbol binario de búsqueda en el cual se debe cumplir la siguiente condición: Para todo nodo T del árbol, la altura de los subárboles izquierdo y derecho no debe diferir en más de una unidad. Estos árboles también reciben el nombre de AVL en honor a sus inventores, dos matemáticos rusos, G.M.Adelson-Velski y E.M.Landis. la idea de estos árboles es realizar reacomodos o balanceos después de inserciones o eliminaciones de elementos.

Inserción en árboles balanceados

Al insertar un elemento en un árbol balanceado se deben seguir los casos:

1. Las ramas izquierdo (RI) y derecho (RD) del árbol tienen la misma altura (HRI = HRD), por tanto:
   - si se inserta un elemento en RI entonces HRI será mayor a HRD
   - si se inserta un elemento en RD entonces HRD será mayor a HRI

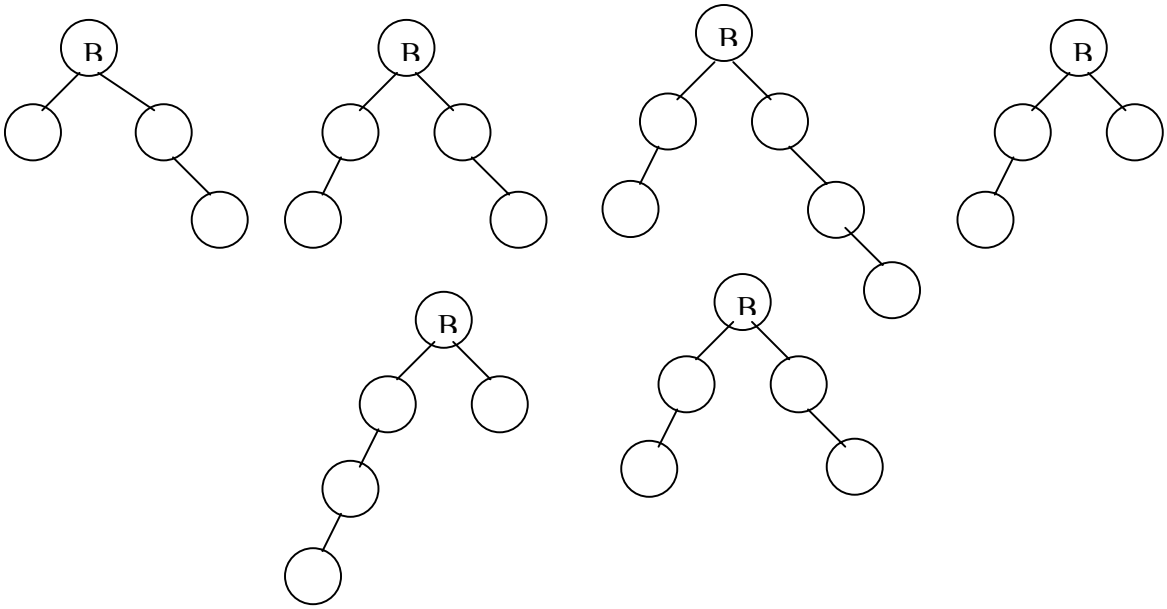


2.- Las ramas izquierdas (RI) y derechas (RD) del árbol tienen diferente altura (HRI =/= HRD). Supóngase que HRI < HRD

- Si se inserta un elemento en la rama izquierda RI entonces HRI = HRD {las alturas de ambas ramas son iguales por lo que se mejora el equilibrio}
- Si se inserta en RD, entonces se rompe el equilibrio y será necesario reestructurarlo.

Supongase HRI > HRD :

- Si se inserta en RI, entonces se rompe el criterio de equilibrio y será necesario reestructurar el árbol.
- Si se inserta un elemento en RD, entonces HRD = HRI. Por lo que las ramas tendrán la misma altura mejorando el equilibrio del árbol.



Borrado en árboles balanceados

Consiste en quitar un nodo sin violar los principios del árbol balanceado. La operación de borrado debe distinguir los siguientes casos:

1. Si el elemento a borrar es terminal u hoja, simplemente se suprime.
2. Si el elemento a borrar tiene un solo descendiente entonces, tiene que sustituirse por ese descendiente.
3. Si el elemento a borrar tiene dos descendientes, entonces tiene que sustituirse por el nodo que se encuentra mas a la izquierda en el subárbol o por el nodo que se encuentra mas a la derecha en el subárbol izquierdo.

Para poder determinar si un árbol esta balanceado se debe calcular su factor de equilibrio de un nodo (FE) que se define como:

La altura del subárbol derecho menos la altura del subárbol izquierdo.

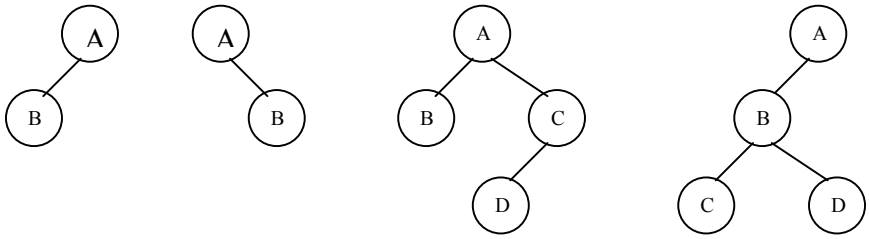
FE = HRD - HRI

Árboles Binarios

En un árbol binario cada nodo puede tener como máximo dos subárboles; y siempre es necesario distinguir entre el subárbol izquierdo y el derecho. Se define un árbol binario de tipo T como una estructura homogénea que es la concatenación de un elemento de tipo T llamado raíz, con dos árboles binarios disjuntos, llamados subárbol izquierdo y derecho. Los árboles binarios también se conocen como árboles ordenados de grado dos. Un árbol ordenado es aquel en el las ramas de los nodos del árbol están ordenadas.

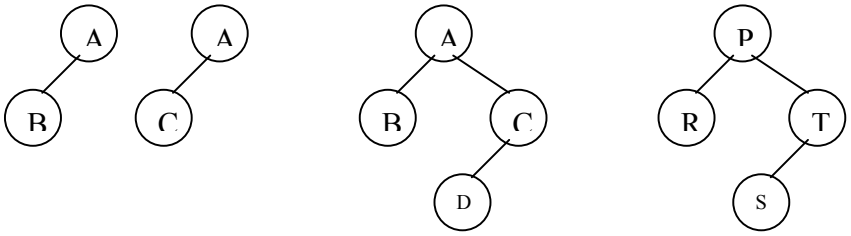
Árboles Binarios Distintos

Dos árboles binarios son distintos cuando sus estructuras son diferentes.



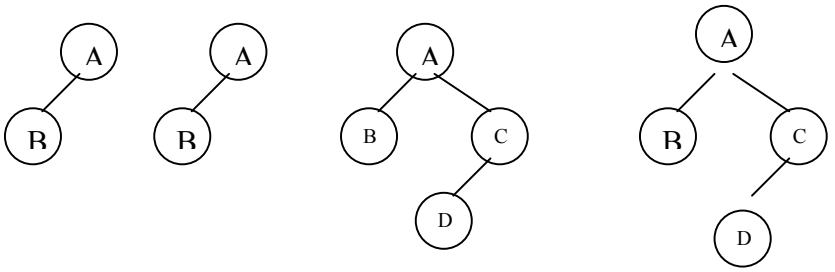
Árboles Binarios Similares

Dos árboles binarios son similares cuando sus estructuras son idénticas, pero la información que contienen en sus nodos difiere entre sí.



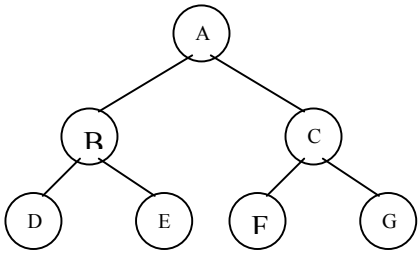
Árboles Binarios Equivalentes

Los árboles binarios equivalentes se definen como aquellos que son similares y además los nodos contienen la misma información.



Árboles Binarios Completos

Se define un árbol binario completo como un árbol en el que todos sus nodos, excepto los del ultimo nivel, tienen dos hijos; el subárbol izquierdo y el derecho.



Recorridos en Árboles Binarios

Recorrer significa visitar los nodos del árbol en forma sistemática; de tal manera que todos los nodos sean visitados una sola vez. Existen tres formas diferentes de efectuar un recorrido.

En preorden.

- Visitar la raíz.
- Recorrer el subárbol izquierdo.
- Recorrer el subarbol derecho.

En inorden.

- Recorrer el subarbol izquierdo.
- Visitar la raíz.
- Recorrer el subarbol derecho.

En postorden.

- Recorrer el subarbol izquierdo.
- Recorrer el subarbol derecho.
- Visitar la raíz.

Algoritmos.

Preorden (nodo)

{Recorrido preorden en un árbol binario. Nodo es un dato de tipo puntero} {INFO, IZQ., DER. Son campos del resgistro nodo. INFO es una variable de tipo carácter IZQ. Y DER. Son variables de tipo puntero}

1. si NODO =/= NIL entonces
   visitar el nodo { NODO^.INFO}

regresar a preorden con NODO^.IZQ.
{llamada recursiva a preorden con la rama izquierda del nodo en cuestión}
regresar a preorden con NODO^.DER.
{llamada recursiva a preorden con la rama derecha del nodo en cuestión}
2. {fin del condicional del paso 1}

Arboles m-arios

W.I. Landauer sugirió la construcción de un árbol m-ario obligando que el nivel L estuviera casi completamente lleno antes de que apareciera nada en el nivel L +1;Landauer supuso que teníamos que buscar elementos en el árbol mas a menudo de lo que necesitábamos insertarlos o eliminarlos. Un árbol B de orden m es un árbol que satisface las siguiente propiedades:

i) Cada nodo tiene <= m hijos
ii) Cada nodo, excepto la raíz y las hojas, tienen >= m/2 hijos
iii) La raíz tiene por lo menos dos hijos
iv) Todas las hojas aparecen en el mismo nivel y no llevan información
v) Un nodo no hoja con k hijos contiene k-1 claves.

- ***Montículos (heaps), balanceados, árboles-AVL***

Montículos binarios (heaps)

Veamos otro tipo especial de árbol binario, los llamados heaps (montículos), que se pueden representar eficazmente con un vector. Definición: un montículo de máximos (mínimos), o max heap (min heap), es un árbol binario completo tal que, el valor de la clave de cada nodo es mayor (menor) o igual que las claves de sus nodos hijos (si los tiene). De la definición se deduce que la clave contenida en el nodo raíz de un montículo de máximos es la mayor clave del árbol, pero esto no quiere decir que sea la única con ese valor. Análogamente sucede con los montículos de mínimos y la menor clave del árbol. La estructura del montículo tiene interesantes aplicaciones, por ejemplo, la ordenación de arrays (algoritmo heapsort) o el mantenimiento de las llamadas colas de prioridad. Las colas de prioridad son un tipo especial de colas donde todo elemento que se inserta en la cola lleva asociado una prioridad, de forma que el elemento que se borra de la cola es el primero entre los que tengan la máxima prioridad (en montículos de máximos). Los montículos, como árboles binarios completos que son, se pueden representar de forma eficaz mediante una estructura secuencial (un array). A cada nodo del árbol se le asigna una posición dentro de la secuencia en función del nivel del árbol en el que se encuentra y dentro de ese nivel de la posición ocupada de izquierda a derecha. De forma que para representado por el elemento A[i] se cumplen las propiedades que vimos al principio del tema para árboles completos, es decir, el nodo padre estará localizado en A[i div 2], si i>1, y los hijos estaran localizados en A[2i] y A[2i+1]. Una característica fundamental de esta estructura de datos es que la propiedad que caracteriza a un montículo puede ser restaurada eficazmente tras cualquier modificación de un nodo (cambio de su valor, inserción, borrado, etc). Supongamos que se trata de un montículo de máximos (cualquier comentario sobre este tipo de montículos se puede extender fácilmente a los montículos de mínimos). En este caso, se debe cumplir que la clave de cualquier nodo sea mayor o igual que las claves de sus hijos. Si se modifica el valor de un nodo incrementando su valor, entonces es posible que la propiedad del montículo no se cumpla con respecto a sus nodos antecesores, no respecto a sus hijos. Esto implicaría ir ascendiendo por el árbol comprobando si el nodo modificado es mayor que el padre, si es así intercambiarlos y repetir el proceso en un nivel superior hasta que se encuentre que no es necesario realizar ningún intercambio o que se llegue al nodo raíz. En cualquier caso, tras ese proceso se habrá restaurado la propiedad del montículo. Si por el contrario, modificamos un nodo disminuyendo su valor, el problema consistirá en comprobar la propiedad respecto a sus descendientes. Si la clave del nodo es mayor que las claves de sus hijos, entonces la propiedad se sigue cumpliendo, si no es así, hay que intercambiar la clave de este nodo con la del hijo que tenga la clave máxima y repetir todo el proceso desde el principio con este nodo hijo hasta que no se realice el intercambio. Estas dos posibilidades de restauración del montículo se pueden expresar en forma de los dos siguientes algoritmos:

```
Algoritmo Subir
  Entradas
   p: arbol[1..n]
   i: indice
  Inicio
   si (i > 1) Y (A[i] > A[i div 2]) entonces
      intercambiar(A[i], A[i div 2])
      Subir(A, i div 2)
   fin_si
```

```
  Fin

Algoritmo Bajar
  Entradas
   p: arbol[1..n]
   i: indice
  Variable
   max: indice
  Inicio
   max <-- i
   si (2i <= n) Y (A[2i] > A[max]) entonces
      max <-- (2i)
   si (2i + 1 <= n) Y (A[2i + 1] > A[max]) entonces
      max <-- (2i + 1)
    si (max <> i) entonces
      intercambiar(A[i], A[max])
      Bajar(A, max)
   fin_si
  Fin
```

A partir de estos dos algoritmos básicos que permiten restaurar la propiedad del montículo, se pueden definir las operaciones de inserción y borrado de elementos en esta estructura de datos.

Inserción

Al intentar añadir un nuevo elemento al montículo, no se sabe cuál será la posición que ocupará el nuevo elemento de la estructura, pero lo que si se sabe es que, por tratarse de un árbol completo, dónde se debe enlazar el nuevo nodo, siempre será en la última posición de la estructura. Por ejemplo, en el siguiente montículo formado por cinco elementos, si se intenta añadir un nuevo elemento, sea el que sea, se conoce cuál será la estructura final del árbol: Si se almacena en el nuevo nodo la información que se desea insertar, lo más probable será que la propiedad del montículo no se conserve. Como el nuevo nodo siempre es terminal, para mantener la propiedad del montículo bastará con aplicar el algoritmo Subir a partir del nodo insertado. Con esto, el algoritmo de inserción resulta verdaderamente simple:

```
Algoritmo Insertar
  Entradas
   A: arbol[1..n]
   x: Valor
  Inicio
   si (A.num < MAX_NODOS) entonces
      A.num <-- A.num + 1
      A[n] <-- x
      Subir(A, n)
   fin_si
  Fin
```

Borrado

Al igual que en la operación de inserción, hay que tener en cuenta que el montículo es un árbol completo y que, para que se mantenga esa propiedad, el nodo que debe desaparecer realmente es el último. Por lo tanto, la estrategia a seguir para borrar cualquier nodo del montículo podría ser: sustituir la información del nodo a borrar por la del último nodo del montículo y considerar que el árbol tiene un nodo menos; como esta modificación seguramente habrá alterado la propiedad del montículo, entonces aplicar el algoritmo Subir o Bajar, lo que sea preciso, para restaurar esa propiedad. La aplicación de cualquiera de los algoritmos dependerá de que la modificación de la información del nodo haya implicado un incremento o disminución de la clave.

```
Algoritmo Borrar
  Entradas
   p: arbol[1..n]
   i: indice
  Variable
   x: Valor
  Inicio
   x <-- A[i]
   A[i] <-- A[n]
```

```
  n <-- n - 1
  si (A[i] <> x) entonces
     si (A[i] > x) entonces Bajar(A, i)
     sino Subir(A, i)
  fin_si
 Fin
```

Aplicación de los montículos

Los montículos tienen como aplicación fundamental el mantenimiento de las llamadas colas de prioridad. Esta estructura de datos es un tipo especial de cola, donde cada elemento lleva asociado un valor de prioridad, de forma que cuando se borra un elemento de la estructura éste será el primero de los elementos con la mayor prioridad. En cualquier instante se puede insertar un elemento con prioridad arbitraria. Si la utilización de la cola de prioridad requiere borrar el elemento con la mayor prioridad, entonces se utiliza para su representación un montículo de máximos, donde resulta inmediata la localización de ese elemento. De forma análoga se haría si necesitamos borrar el elemento con la menor prioridad, utilizaríamos un montículo de mínimos. En un montículo de máximos resulta inmediato obtener el valor máximo de las claves del árbol. Como se ha visto anteriormente, este valor está siempre situado en la raíz del árbol, por lo tanto bastaría con acceder al primer elemento del array A[1].

```
Algoritmo EliminarMaximo
  Entradas
   p: arbol[1..n]
  Inicio
   si (p.num = 0) entonces "Error: arbol vacío."
   A[i] <-- A[n]
   n <-- n - 1
   si (A[i] <> x) entonces
      si (A[i] > x) entonces Bajar(A, i)
      sino Subir(A, i)
   fin_si
  Fin
```

- ***Gráficas: no dirigidas, dirigidas, multigráficas, pesadas***

Grafos

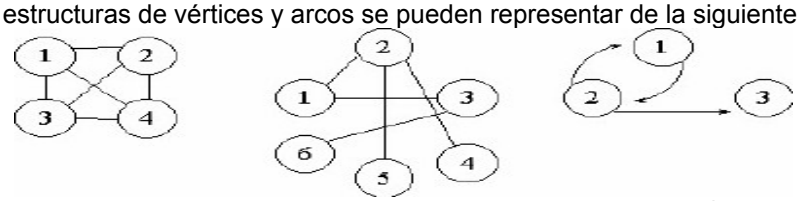
Fundamentos y terminología básica

Un grafo, G, es un par, compuesto por dos conjuntos V y A. Al conjunto V se le llama conjunto de vértices o nodos del grafo. A es un conjunto de pares de vértices, estos pares se conocen habitualmente con el nombre de arcos o ejes del grafo. Se suele utilizar la notacion G = (V, A) para identificar un grafo. Los grafos representan un conjunto de objetos donde no hay restricción a la relación entre ellos. Son estructuras más generales y menos restrictivas. Podemos clasificar los grafos en dos grupos: dirigidos y no dirigidos. En un grafo no dirigido el par de vértices que representa un arco no está ordenado. Por lo tanto, los pares (v1, v2) y (v2, v1) representan el mismo arco. En un grafo dirigido cada arco está representado por un par ordenado de vértices , de forma que y representan dos arcos diferentes. Ejemplos de grafos (dirigidos y no dirigidos):

G1 = (V1, A1)
V1 = {1, 2, 3, 4} A1 = {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}
G2 = (V2, A2)
V2 = {1, 2, 3, 4, 5, 6} A2 = {(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)}

G3 = (V3, A3)
V3 = {1, 2, 3} A3 = { <1, 2>, <2, 1>, <2, 3> }

Gráficamente estas tres estructuras de vértices y arcos se pueden representar de la siguiente manera:



Los grafos permiten representar conjuntos de objetos arbitrariamente relacionados. Se puede asociar el conjunto de vértices con el conjunto de objetos y el conjunto de arcos con las relaciones que se establecen entre ellos. Los grafos son modelos matemáticos de numerosas situaciones reales: un mapa de carreteras, la red de ferrocarriles, el plano de un circuito eléctrico, el esquema de la red telefónica de una compañia, etc. El número de distintos pares de vértices (v(i), v(j)), con v(i) <> v(j), en un grafo con n vértices es n*(n-1)/2. Este es el número máximo de arcos en un grafo no

dirigido de n vértices. Un grafo no dirigido que tenga exactamente n*(n-1)/2 arcos se dice que es un grafo completo. En el caso de un grafo dirigido de n vértices el número máximo de arcos es n*(n-1).
Algunas definiciones básicas en grafos:

Orden de un grafo: es el número de nodos (vértices) del grafo.
Grado de un nodo: es el número de ejes (arcos) que inciden sobre el nodo
Grafo simétrico: es un grafo dirigido tal que si existe la relación entonces existe , con u, v pertenecientes a V.
Grafo no simétrico: es un grafo que no cumple la propiedad anterior.
Grafo reflexivo: es el grafo que cumple que para todo nodo u de V existe la relación (u, u) de A.
Grafo transitivo: es aquél que cumple que si existen las relaciones (u, v) y (v, z) de A entonces existe (u, z) de A.
Grafo completo: es el grafo que contiene todos los posibles pares de relaciones, es decir, para cualquier par de nodos u, v de V, (u <> v), existe (u, v) de A.
Camino: un camino en el grafo G es una sucesión de vértices y arcos: v(0), a(1), v(1), a(2), v(2), ... , a(k), v(k); tal que los extremos del arco a(i) son los vértices v(i-1) y v(i).
Longitud de un camino: es el número de arcos que componen el camino.
Camino cerrado (circuito): camino en el que coinciden los vértices extremos (v(0) = v(k)).
Camino simple: camino donde sus vértices son distintos dos a dos, salvo a lo sumo los extremos.
Camino elemental: camino donde sus arcos son distintos dos a dos.
Camino euleriano: camino simple que contiene todos los arcos del grafo.
Grafo euleriano: es un grafo que tiene un camino euleriano cerrado.
Grafo conexo: es un grafo no dirigido tal que para cualquier par de nodos existe al menos un camino que los une.
Grafo fuertemente conexo: es un grafo dirigido tal que para cualquier par de nodos existe un camino que los une.
Punto de articulación: es un nodo que si desaparece provoca que se cree un grafo no conexo.
Componente conexa: subgrafo conexo maximal de un grafo no dirigido (parte más grande de un grafo que sea conexa).

Representación de grafos

Existen tres maneras básicas de representar los grafos: mediante matrices, mediante listas y mediante matrices dispersas. Cada representación tiene unas ciertas ventajas e inconvenientes respecto de las demás, que comentaremos más adelante.

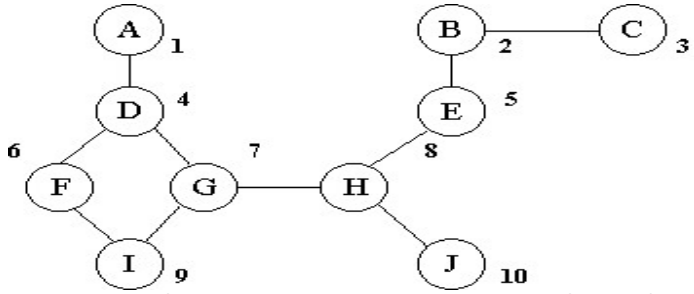
Representación mediante matrices: matrices de adyacencia

Un grafo es un par compuesto por dos conjuntos: un conjunto de nodos y un conjunto de relaciones entre los nodos. La representación tendrá que ser capaz de guardar esta información en memoria. La forma más fácil de guardar la información de los nodos es mediante la utilización de un vector que indexe los nodos, de manera que los arcos entre los nodos se pueden ver como relaciones entre los índices. Esta relación entre índices se puede guardar en una matriz, que llamaremos de adyacencia.

```
Const
  MAX_NODOS = ??;
Type
  Indice = 1..MAX_NODOS;
  Valor_Nodo = ??;
  Valor_Arco = ??;
  Arco = Record
           Info: Valor_Arco;       (* información asociada a cada arco *)
           Existe: Boolean;
         end;
  Nodo = Record
           Info: Valor_Nodo;    (* información asociada a cada nodo *)
           Existe: Boolean;
         end;
  Grafo = Record
            Nodos: Array[Indice] of Nodo;
            Arcos: Array[Indice, Indice] of Arco;
          end;
```

Con esta representación tendremos que reservar al menos del orden de (n^2) espacios de memoria para la información de los arcos, y las operaciones relacionadas con el grafo implicarán, habitualmente, recorrer toda la matriz, con lo que el orden de las operaciones será, en general, cuadrático, aunque tengamos un número de

relaciones entre los nodos mucho menor que (n^2). En cambio, con esta representación es muy fácil determinar, a partir de dos nodos, si están o no relacionados: sólo hay que acceder al elemento adecuado de la matriz y comprobar el valor que guarda. Ejemplo:



Supongamos el grafo representado en la figura anterior. A partir de ese grafo la información que guardariamos, con esta representación, sería:

GRAFO

Nodos

	1	2	3	4	5	6	7	8	9	10	11	12
Existe	T	T	T	T	T	T	T	T	T	T	F	F
	A	B	C	D	E	F	G	H	I	J		

Arcos

Existe	1	2	3	4	5	6	7	8	9	10	11	12
1	F	F	F	V	F	F	F	F	F	F		
2	F	F	F	F	V	F	F	F	F	F		
3	F	F	F	F	V	F	F	F	F	F		
4	V	F	F	F	F	V	V	F	F	F		
5	F	V	V	F	F	F	F	V	F	V		
6	F	F	F	V	F	F	V	F	F	F		
7	F	F	F	V	F	F	F	V	V	F		
8	F	F	F	F	V	F	V	F	F	V		
9	F	F	F	F	F	V	V	F	F	F		
10	F	F	F	F	V	F	F	V	F	F		
11												
12												

Representación mediante punteros: listas de adyacencia

En las listas de adyacencia se intenta evitar justamente el reservar espacio para aquellos arcos que no contienen ningún tipo de información. El sustituto obvio a los vectores con huecos son las listas. En las listas de adyacencia lo que haremos será guardar por cada nodo, además de la información que pueda contener el propio nodo, una lista dinámica con los nodos a los que se puede acceder desde él. La información de los nodos se puede guardar en un vector, al igual que antes, o en otra lista dinámica. Si elegimos la representación en un vector para los nodos, tendríamos la siguiente definición de grafo en Pascal:

```
Const
  MAX_NODOS = ??;
Type
  Indice = 1..MAX_NODOS;
  Valor_Nodo = ??;
  Valor_Arco = ??;
  Punt_Arco = ^Arco;
  Arco = Record
           Info: Valor_Arco;       (* información asociada a cada arco *)
           Destino: Indice;
           Sig_Arco: Punt_Arco;
         end;
  Nodo = Record
           Info: Valor_Nodo;    (* información asociada a cada nodo *)
           Existe: Boolean;
           Lista_Arcos: Punt_Arco;
         end;
  Grafo = Record
            Nodos: Array[Indice] of Nodo;
          end;
```

En general se está guardando menor cantidad de elementos, sólo se reservará memoria para aquellos arcos que efectivamente existan, pero como contrapartida estamos guardando más espacio para cada uno de los arcos (estamos añadiendo el índice destino del arco y el puntero al siguiente elemento de la lista de arcos). Las tareas relacionadas con el recorrido del grafo supondrán sólo trabajar con los vértices existentes en el grafo, que puede ser mucho menor que (n^2). Pero comprobar las relaciones entre nodos no es tan directo como lo era en la matriz, sino

que supone recorrer la lista de elementos adyacentes perteneciente al nodo analizado. Además, sólo estamos guardando realmente la mitad de la información que guardábamos en el caso anterior, ya que las relaciones inversas (las relaciones que llegan a un cierto nodo) en este caso no se guardan, y averiguarlas supone recorrer todas las listas de todos los nodos.

Representación mediante punteros: matrices dispersas

Para evitar uno de los problemas que teníamos con las listas de adyacencia, que era la dificultad de obtener las relaciones inversas, podemos utilizar las matrices dispersas, que contienen tanta información como las matrices de adyacencia, pero, en principio, no ocupan tanta memoria como las matrices, ya que al igual que en las listas de adyacencia, sólo representaremos aquellos enlaces que existen en el grafo.

```
Const
  MAX_NODOS = ??;
Type
  Indice = 1..MAX_NODOS;
  Valor_Nodo = ??;
  Valor_Arco = ??;
 Punt_Arco = ^Arco;
  Arco = Record
            Info: Valor_Arco;       (* información asociada a cada arco *)
            Origen: Indice;
            Destino: Indice;
            Sig_Arco_Salida: Punt_Arco;
            Sig_Arco_Entrada: Punt_Arco;
          end;
 Nodo = Record
            Info: Valor_Nodo;    (* información asociada a cada nodo *)
            Existe: Boolean;
            Lista_Arcos_Salida: Punt_Arco;
            Lista_Arcos_Entrada: Punt_Arco;
          end;
  Grafo = Record
             Nodos: Array[Indice] of Nodo;
             N_Nod: 0..MAX_NODOS;
             Existe_Nodo: Array[Indice] of Boolean;
           end;
```

Recorrido de grafos

Recorrer un grafo supone intentar alcanzar todos los nodos que estén relacionados con uno dado que tomaremos como nodo de salida. Existen básicamente dos técnicas para recorrer un grafo: el recorrido en anchura; y el recorrido en profundidad.

Recorrido en anchura o BFS (Breadth First Search)

El recorrido en anchura supone recorrer el grafo, a partir de un nodo dado, en niveles, es decir, primero los que están a una distancia de un arco del nodo de salida, después los que están a dos arcos de distancia, y así sucesivamente hasta alcanzar todos los nodos a los que se pudiese llegar desde el nodo salida.

El algoritmo general de recorrido es el siguiente:

```
Algoritmo Recorrido_en_Anchura (BFS)
  Entradas
   gr: Grafo                (* grafo a recorrer *)
   nodo_salida: Indice    (* origen del recorrido *)
   Variables
   queue: Cola de Indice
   aux_nod1, aux_nod2: Indice
  Inicio
  Iniciar_Cola(queue)
   Procesar(nodo_salida)
   Visitado(nodo_salida) <-- CIERTO
   Encolar(queue, nodo_salida)
   mientras NO Cola_Vacia(queue) hacer
      aux_nod1 <-- Desencolar(queue)
           para (todos los nodos), aux_nod2, adyacentes a aux_nod1 hacer
```

```
            si NO Visitado[aux_nod2] entonces
           Procesar(aux_nod2)
                  Visitado[aux_nod2] <-- CIERTO
                  Encolar(queue, aux_nod2)
            fin_si
          fin_para
  fin_mientras
 Fin
```

La diferencia a la hora de implementar el algoritmo general para cada una de las implementaciones de la estructura de datos grafo, residirá en la manera de averiguar los diferentes nodos adyacentes a uno dado. En el caso de las matrices de adyacencia se tendrán que comprobar si los enlaces entre los nodos existen en la matriz. En los casos de las listas de adyacencia y de las matrices dispersas sólo habrá que recorrer las listas de enlaces que parten del nodo en cuestión para averiguar qué nodos son adyacentes al estudiado.

Recorrido en profundidad o DFS (Depth First Search)

A diferencia del algoritmo anterior, el recorrido en profundidad trata de buscar los caminos que parten desde el nodo de salida hasta que ya no es posible avanzar más. Cuando ya no puede avanzarse más sobre el camino elegido, se vuelve atrás en busca de caminos alternativos, que no se estudiaron previamente. El algoritmo es similar al anterior, pero utilizando, para guardar los nodos accesibles desde uno dado, una pila en lugar de una cola.

```
Algoritmo Recorrido_en_Profundidad (DFS)
  Entradas
   gr: Grafo                (* grafo a recorrer *)
   nodo_salida: Indice      (* origen del recorrido *)
  Variables

   stack: Pila de Indice
   aux_nod1, aux_nod2: Indice
  Inicio
   Iniciar_Pila(stack)
   Procesar(nodo_salida)
   Visitado(nodo_salida) <-- CIERTO
   Apilar(stack, nodo_salida)
   mientras NO Pila_Vacia(stack) hacer
      aux_nod1 <-- Desapilar(stack)
           para (todos los nodos), aux_nod2, adyacentes a aux_nod1 hacer
              si NO Visitado[aux_nod2] entonces
             Procesar(aux_nod2)
                    Visitado[aux_nod2] <-- CIERTO
                    Apilar(stack, aux_nod2)
              fin_si
           fin_para
  fin_mientras
Fin
```

La utilización de la pila se puede sustituir por la utilización de la recurrencia, de manera que el algoritmo quedaría como sigue:

```
Algoritmo Recorrido_en_Profundidad (DFS)
  Entradas
   gr: Grafo                (* grafo a recorrer *)
   nodo_salida: Indice      (* origen del recorrido *)
  Variables
   aux_nod2: Indice
  Inicio
   Procesar(nodo_salida)
   Visitado(nodo_salida) <-- CIERTO
  para (todos los nodos), aux_nod2, adyacentes a aux_nod1 hacer
      si NO Visitado[aux_nod2] entonces
            Recorrido_en_Profundidad(gr, aux_nod2)
           fin_si
```

```
  fin_para
Fin
```

II. Archivos

1. Archivos de almacenamiento

- ***Dispositivos de almacenamiento***

INTRODUCCIÓN.

Debido a la cantidad de información que manejamos actualmente, los dispositivos de almacenamiento se han vuelto casi tan importantes como el mismísimo computador. Aunque actualmente existen dispositivos para almacenar que superan las 650 MB de memoria, aún seguimos quejándonos por la falta de capacidad para transportar nuestros documentos y para hacer Backups de nuestra información más importante. Todo esto sucede debido al aumento de software utilitario que nos permite, por dar un pequeño ejemplo, convertir nuestros Cds en archivos de Mp3. El espacio en nuestro Disco duro ya no es suficiente para guardar tal cantidad de información; por lo que se nos es de urgencia conseguir un medo alternativo de almacenamiento para guardar nuestros Cds en Mp3 o los programas que descargamos de Internet.

La tecnología óptica

La tecnología óptica de almacenamiento por láser es bastante más reciente. Su primera aplicación comercial masiva fue el superexitoso CD de música, que data de comienzos de la década de 1.980. Los fundamentos técnicos que se utilizan son relativamente sencillos de entender: un haz láser va leyendo (o escribiendo) microscópicos agujeros en la superficie de un disco de material plástico, recubiertos a su vez por una capa transparente para su protección del polvo. Realmente, el método es muy similar al usado en los antiguos discos de vinilo, excepto porque la información está guardada en formato digital (unos y ceros como valles y cumbres en la superficie del CD) en vez de analógico y por usar un láser como lector. El sistema no ha experimentado variaciones importantes hasta la aparición del DVD, que tan sólo ha cambiado la longitud de onda del láser, reducido el tamaño de los agujeros y apretado los surcos para que quepa más información en el mismo espacio. **Disco de vídeo digital**, también conocido en la actualidad como disco versátil digital (DVD), un dispositivo de almacenamiento masivo de datos cuyo aspecto es idéntico al de un disco compacto, aunque contiene hasta 25 veces más información y puede transmitirla al ordenador o computadora unas 20 veces más rápido que un CD-ROM. Su mayor capacidad de almacenamiento se debe, entre otras cosas, a que puede utilizar ambas caras del disco y, en algunos casos, hasta dos capas por cada cara, mientras que el CD sólo utiliza una cara y una capa. Las unidades lectoras de DVD permiten leer la mayoría de los CDs, ya que ambos son discos ópticos; no obstante, los lectores de CD no permiten leer DVDs. En un principio se utilizaban para reproducir películas, de ahí su denominación original de disco de vídeo digital. Hoy, los DVD-Vídeo son sólo un tipo de DVD que almacenan hasta 133 minutos de película por cada cara, con una calidad de vídeo LaserDisc y que soportan sonido digital Dolby surround; son la base de las instalaciones de cine en casa que existen desde 1996. Además de éstos, hay formatos específicos para la computadora que almacenan datos y material interactivo en forma de texto, audio o vídeo, como los DVD-R, unidades en las que se puede grabar la información una vez y leerla muchas, DVD-RW, en los que la información se puede grabar y borrar muchas veces, y los DVD-RAM, también de lectura y escritura. En 1999 aparecieron los DVD-Audio, que emplean un formato de almacenamiento de sonido digital de segunda generación con el que se pueden recoger zonas del espectro sonoro que eran inaccesibles al CD-Audio. Todos los discos DVD tienen la misma forma física y el mismo tamaño, pero difieren en el formato de almacenamiento de los datos y, en consecuencia, en su capacidad. Así, los DVD-Vídeo de una cara y una capa almacenan 4,7 GB, y los DVD-ROM de dos caras y dos capas almacenan hasta 17 GB. Del mismo modo, no todos los DVDs se pueden reproducir en cualquier unidad lectora; por ejemplo, un DVD-ROM no se puede leer en un DVD-Vídeo, aunque sí a la inversa. Por su parte, los lectores de disco compacto, CD, y las unidades de DVD, disponen de un láser, ya que la lectura de la información se hace por procedimientos ópticos. En algunos casos, estas unidades son de sólo lectura y en otros, de lectura y escritura.

Disco duro

Disco duro, en los ordenadores o computadoras, unidad de almacenamiento permanente de gran capacidad. Está formado por varios discos apilados —dos o más—, normalmente de aluminio o vidrio, recubiertos de un material ferromagnético. Como en los disquetes, una cabeza de lectura/escritura permite grabar la información, modificando las propiedades magnéticas del material de la superficie, y leerla posteriormente (La tecnología magnética, consiste en la aplicación de campos magnéticos a ciertos materiales cuyas partículas reaccionan a esa influencia, generalmente orientándose en unas determinadas posiciones que conservan tras dejar de aplicarse el campo magnético. Esas posiciones representan los datos, bien sean una canción, bien los bits que forman una imagen o un documento importante.); esta operación se puede hacer un gran número de veces. La mayor parte de los discos duros son fijos, es decir, están alojados en el ordenador de forma permanente. Existen también discos duros removibles, como los discos Jaz de Iomega, que se utilizan generalmente para hacer backup —copias de seguridad de los discos duros— o para transferir grandes cantidades de información de un ordenador a otro. El primer disco duro se instaló en un

ordenador personal en 1979; era un Seagate con una capacidad de almacenamiento de 5 MB. Hoy día, la capacidad de almacenamiento de un disco duro puede superar los 50 MB. A la vez que aumentaba la capacidad de almacenamiento, los discos duros reducían su tamaño; así se pasó de las 12 pulgadas de diámetro de los primeros, a las 3,5 pulgadas de los discos duros de los ordenadores portátiles o las 2,5 pulgadas de los discos de los *notebooks* (ordenadores de mano). Modernamente, sólo se usan en el mundo del PC dos tipos de disco duro: el IDE y el SCSI (leído "escasi"). La diferencia entre estos Discos duros radica en la manera de conectarlos a la MainBoard.

IDE

Los discos IDE son los más habituales; ofrecen un rendimiento razonablemente elevado a un precio económico y son más o menos fáciles de instalar. Sin embargo, se ven limitados a un número máximo de 4 dispositivos (y esto con las controladoras EIDE, las IDE originales sólo pueden manejar 2).

Su conexión se realiza mediante un cable plano con conectores con 40 pines colocados en dos hileras (aparte del cable de alimentación, que es común para todos los tipos de disco duro). Así pues, para identificar correctamente un disco IDE basta con observar la presencia de este conector, aunque para estar seguros al 100% deberemos buscar unos microinterruptores ("jumpers") que, en número de 2 a 4, permiten elegir el orden de los dispositivos (es decir, si se comportan como "Maestro" o como "Esclavo").

SCSI

Esta tecnología es mucho menos utilizada, pero no por ser mala, sino por ser relativamente cara. Estos discos suelen ser más rápidos a la hora de transmitir datos, a la vez que usan menos al procesador para hacerlo, lo que se traduce en un aumento de prestaciones. Es típica y casi exclusiva de ordenadores caros, servidores de red y muchos Apple Macintosh.

Los conectores SCSI son múltiples, como lo son las variantes de la norma: SCSI-1, SCSI-2, Wide SCSI, Ultra SCSI... Pueden ser planos de 50 contactos en 2 hileras, o de 68 contactos, o no planos con conector de 36 contactos, con mini-conector de 50 contactos...

Una pista para identificarlos puede ser que, en una cadena de dispositivos SCSI (hasta 7 ó 15 dispositivos que van intercalados a lo largo de un cable o cables, como las bombillas de un árbol de Navidad), cada aparato tiene un número que lo identifica, que en general se puede seleccionar. Para ello habrá una hilera de jumpers, o bien una rueda giratoria, que es lo que deberemos buscar.

MFM, ESDI

Muy similares, especialmente por el hecho de que están descatalogados. Su velocidad resulta insufrible, más parecida a la de un disquete que a la de un disco duro moderno. Se trata de cacharros pesados, de formato casi siempre 5,25 pulgadas, con capacidades de 10, 20, 40 o hasta 80 megas máximo.

Dispositivos Periféricos.

Jaz (Iomega) - 1 GB ó 2 GB

- **Pros:** capacidad muy elevada, velocidad, portabilidad
- **Contras:** inversión inicial, no tan resistente como un magneto-óptico, cartuchos relativamente caros

Las cifras de velocidad del Jaz son absolutamente alucinantes, casi indistinguibles de las de un disco duro moderno: poco más de 5 MB/s y menos de 15 ms. La razón de esto es fácil de explicar: cada cartucho Jaz es internamente, a casi todos los efectos, un disco duro al que sólo le falta el elemento lector-grabador, que se encuentra en la unidad. Por ello, atesora las ventajas de los discos duros: gran capacidad a bajo precio y velocidad, junto con sus inconvenientes: información sensible a campos magnéticos, durabilidad limitada en el tiempo, relativa fragilidad. De cualquier forma, y sin llegar a la extrema resistencia de los discos Zip, podemos calificar este soporte de *duro* y fiable, aunque la información nunca estará tan a salvo como si estuviera guardada en un soporte óptico o magneto-óptico.

Aplicaciones

Almacenamiento masivo de datos que deben guardarse y recuperarse con la mayor velocidad posible, lo cual lo hace ideal para la edición de vídeo digital (casi una hora en formato MPEG); en general, sirve para lo mismo que los discos duros, pero con la ventaja de su portabilidad y fácil almacenaje. En cuanto a defectos y críticas, aparte de que los datos no duren "para siempre", sólo tiene un inconveniente: el precio. La unidad lectora-grabadora de 1 GB vale una respetable cantidad de dinero, unos $650.000, y los discos unos $180.000 c/u.

Zip (Iomega) - 100 MB

Pros: portabilidad, reducido formato, precio global, muy extendido

Contras: capacidad reducida, incompatible con disquetes de 3,5"

Las unidades Zip se caracterizan externamente por ser de un color azul oscuro, al igual que los disquetes habituales (los hay de todos los colores). Estos discos son dispositivos magnéticos un poco mayores que los clásicos disquetes de 3,5 pulgadas, aunque mucho más robustos y fiables, con una capacidad sin compresión de 100 MB una vez formateados. Su capacidad los hace inapropiados para hacer copias de seguridad del disco duro completo, aunque perfectos para archivar todos los archivos referentes a un mismo tema o proyecto en un único disco. Su velocidad de transferencia de datos no resulta comparable a la de un disco duro actual, aunque son decenas de veces más rápidos que una disquetera tradicional (alrededor de 1 MB/s). Existen en diversos formatos, tanto internos como externos. Los internos pueden tener interfaz IDE, como la de un disco duro o CD-ROM, o bien SCSI; ambas son bastante rápidas, la

SCSI un poco más, aunque su precio es también superior. Las versiones externas aparecen con interfaz SCSI (con un rendimiento idéntico a la versión interna) o bien conectable al puerto paralelo, sin tener que prescindir de la impresora conectada a éste. El modelo para puerto paralelo pone el acento en la portabilidad absoluta entre ordenadores (Sólo se necesita que tengan el puerto Lpt1) aunque su velocidad es la más reducida de las tres versiones. Muy resistente, puede ser el acompañante ideal de un portátil. Ha tenido gran aceptación, siendo el estándar en su segmento, pese a no poder prescindir de la disquetera de 3,5" con la que no son en absoluto compatibles, aunque sus ventajas puede que suplan este inconveniente. El precio de la versión interna ronda los $262.500 (más IVA) y los Discos alrededor de $35.000 (más IVA). Muchas de las primeras unidades Zip sufrían el denominado "mal del click", que consistía en un defecto en la unidad lectora-grabadora que, tras hacer unos ruiditos o "clicks", destrozaba el disco introducido; afortunadamente, este defecto está corregido en las unidades actuales. En todo caso, los discos son bastante resistentes, pero evidentemente no llegan a durar lo que un CD-ROM.

- ***Parámetros de almacenamiento: números de cilindros, pistas, tiempos de acceso, posicionamiento, densidad de grabación***

Tipos de discos compactos

SOPORTE	CAPACIDAD DE ALMACENAMIENTO	DURACIÓN MÁXIMA DE AUDIO	DURACIÓN MÁXIMA DE VÍDEO	NÚMERO DE CDs A LOS QUE EQUIVALE
Disco compacto (CD)	650 Mb	1 h 18 min.	15 min.	1
DVD una cara / una capa	4,7 Gb	9 h 30 min.	2 h 15 min.	7
DVD una cara / doble capa	8,5 Gb	17 h 30 min.	4 h	13
DVD doble cara / una capa	9,4 Gb	19 h	4 h 30 min.	14
DVD doble cara / doble capa	17 Gb	35 h	8 h	26

- ***Formas de almacenaje: modo texto y modo binario***

Modo texto

En informática, también denominado modo alfanumérico o modo carácter. El modo de operación con el cual algunos equipos informáticos muestran letras y otros caracteres de texto, pero no imágenes gráficas como punteros de *mouse* (ratón) o formato de caracteres de WYSIWYG, es decir, cursivas, números superíndices y otros. Los equipos IBM y compatibles pueden operar tanto en modo texto como en modo gráfico. Los equipos Apple Macintosh, al ser máquinas que se basan en gráficos, operan en modo gráfico.

2. Accesos

- ***Acceso secuencial***

Dependiendo de la manera en que se accesen los registros de un archivo, se le clasifica como SECUENCIAL o como DIRECTO. En el caso de los archivos de ACCESO SECUENCIAL, para tener acceso al registro localizado en la posición N, se deben haber accesado los N-1 registros previos, en un orden secuencial. Cuando se tienen pocos registros en un archivo, o que los registros son pequeños, la diferencia entre los tiempos de acceso de forma secuencial y directa puede no ser perceptible para el usuario; sin embargo, la diferencia viene a ser significativa cuando se manejan archivos con grandes cantidades de información. La forma de manejar los archivos de acceso secuencial es más sencilla en la mayoría de los lenguajes de programación, por lo que su estudio se antepone al de los archivos de acceso directo. El manejo secuencial de un archivo es recomendable cuando se deben procesar todos o la mayoría de los registros, como por ejemplo en los casos de una nómina de empleados o en la elaboración de reportes contables.

- ***Acceso directo y cálculo de posición***

Los archivos directos explotan la capacidad de los discos para acceder directamente a cualquier bloque de dirección conocida. Se requiere de un campo clave en cada registro pero no existe el concepto de ordenamiento secuencial.

Los archivos con tipos están estructurados en elementos o registros (record) cuyo tipo puede ser cualquiera. A los elementos de estos archivos se accede directamente, al no situarse éstos en posiciones físicamente consecutivas, sino en posiciones lógicas. Esta es la razón por la cual se les denomina archivos de acceso aleatorio o directo. Los elementos de los archivos aleatorios son de igual tamaño y el término acceso directo significa que es posible acceder directamente a un elemento con solo especificar su posición.

- ***Acceso por índices: índice secuencial ordenado***

Contienen un área de datos que agrupa a los registros y un área de índice que contiene los niveles de índice. Siempre y cuando el índice sea pequeño se podrá almacenar en la memoria para su rápida búsqueda y acceso.

- ***Acceso por índices: tablas de dispersión***

Se accede indirectamente a los registros por su clave, mediante consulta secuenciales a una tabla que contiene la clave y la dirección relativa de cada registro, y posterior acceso directo al registro.

- ***Acceso por índices: árboles B y B+***

Árboles B y árboles B+

Los árboles B y los árboles B+ son casos especiales de árboles de búsqueda. Un árbol de búsqueda es un tipo de árbol que sirve para guiar la búsqueda de un registro, dado el valor de uno de sus campos. Los índices multinivel de la sección anterior pueden considerarse como variaciones de los árboles de búsqueda. Cada bloque o nodo del índice multinivel puede tener hasta p valores del campo de indexación y p punteros. Los valores del campo de indexación de cada nodo guían al siguiente nodo (que se encuentra en otro nivel), hasta llegar al bloque del fichero de datos que contiene el registro deseado. Al seguir un puntero, se va restringiendo la búsqueda en cada nivel a un subárbol del árbol de búsqueda, y se ignoran todos los nodos que no estén en dicho subárbol. Los árboles de búsqueda difieren un poco de los índices multinivel. Un árbol de búsqueda de orden p es un árbol tal que cada nodo contiene como mucho $p-1$ valores del campo de indexación y p punteros:

$$(P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q)$$

donde $q \leq p$, cada P_i es un puntero a un nodo hijo y cada K_i es un valor de búsqueda proveniente de algún conjunto ordenado de valores. Se supone que todos los valores de búsqueda son únicos. Un árbol de búsqueda debe cumplir, en todo momento, las siguientes restricciones:

1. Dentro de cada nodo: $K_1 < K_2 < \ldots K_{q-1}$.
2. Para todos los valores X del subárbol al que apunta P_i, se tiene: $K_{i-1} < X < K_i$ para $1 < i < q$, $X < K_i$ para $i = 1$ y $K_{i-1} < X$ para $i = q$.

Al buscar un valor X siempre se sigue el puntero P_i apropiado de acuerdo con las condiciones de la segunda restricción. Para insertar valores de búsqueda en el árbol y eliminarlos, sin violar las restricciones anteriores, se utilizan algoritmos que no garantizan que el árbol de búsqueda esté equilibrado (que todas las hojas estén al mismo nivel). Es importante mantener equilibrados los árboles de búsqueda porque esto garantiza que no habrá nodos en niveles muy profundos que requieran muchos accesos a bloques durante una búsqueda. Además, las eliminaciones de registros pueden hacer que queden nodos casi vacíos, con lo que hay un desperdicio de espacio importante que también provoca un aumento en el número de niveles. El árbol B es un árbol de búsqueda, con algunas restricciones adicionales, que resuelve hasta cierto punto los dos problemas anteriores. Estas restricciones adicionales garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar se hacen más complejos para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples, se complican sólo en circunstancias especiales: cuando se intenta insertar en un nodo que está lleno o cuando se intenta borrar en un nodo que está ocupado hasta la mitad. Un árbol B de orden p se define del siguiente modo:

1. La estructura de cada nodo interno tiene la forma:

   $$(P_1, (K_1, Pr_1), P_2, (K_2, Pr_2), P_3, (K_3, Pr_3), \ldots P_{q-1}, (K_{q-1}, Pr_{q-1}), P_q)$$

   donde $q \leq p$. Cada P_i es un puntero a un nodo interno del árbol y cada Pr_i es un puntero al registro del fichero de datos que tiene el valor K_i en el campo de búsqueda o de indexación.
2. Dentro de cada nodo se cumple: $K_1 < K_2 < \ldots K_{q-1}$.
3. Para todos los valores X del campo de indexación del subárbol al que apunta P_i, se cumple: $K_{i-1} < X < K_i$ para $1 < i < q$, $X < K_i$ para $i = 1$ y $K_{i-1} < X$, para $i = q$.
4. Cada nodo tiene, como mucho, p punteros a nodos del árbol.

5. Cada nodo, excepto la raíz y las hojas, tiene, al menos, $\lceil p/2 \rceil$ punteros a nodos del árbol. El nodo raíz tiene, como mínimo, dos punteros a nodos del árbol, a menos que sea el único nodo del árbol.

6. Un nodo con q punteros a nodos, $q \leq p$, tiene $q - 1$ valores del campo de indexación.

7. Todos los nodos hoja están al mismo nivel. Los nodos hoja tienen la misma estructura que los nodos internos, pero los punteros a nodos del árbol son nulos.

Como se puede observar, en los árboles B todos los valores del campo de indexación aparecen alguna vez en algún nivel del árbol, junto con un puntero al fichero de datos. En un árbol B+ los punteros a datos se almacenan sólo en los nodos hoja del árbol, por lo cual, la estructura de los nodos hoja difiere de la de los nodos internos. Los nodos hoja tienen una entrada por cada valor del campo de indexación, junto con un puntero al registro del fichero de datos. Estos nodos están enlazados para ofrecer un acceso ordenado a los registros a través del campo de indexación. Los nodos hoja de un árbol B+ son similares al primer nivel (nivel base) de un índice. Los nodos internos del árbol B+ corresponden a los demás niveles del índice. Algunos valores del campo de indexación se repiten en los nodos internos del árbol B+ con el fin de guiar la búsqueda. En un árbol B+ de orden p la estructura de un nodo interno es la siguiente:

1. Todo nodo interno es de la forma:

   $(P_1, K_1, P_2, K_2, P_3, K_3, \ldots P_{q-1}, K_{q-1}, P_q)$

   donde $q \leq p$. Cada P_i es un puntero a un nodo interno del árbol.

2. Dentro de cada nodo interno se cumple: $K_1 < K_2 < \ldots K_{q-1}$.

3. Para todos los valores X del campo de indexación del subárbol al que apunta P_i, se cumple: $K_{i-1} < X \leq K_i$ para $1 < i < q$, $X \leq K_i$ para $i = 1$ y $K_{i-1} < X$ para $i = q$.

4. Cada nodo interno tiene, como mucho, p punteros a nodos del árbol.

5. Cada nodo interno, excepto la raíz, tiene, al menos, $\lceil p/2 \rceil$ punteros a nodos del árbol. El nodo raíz tiene, como mínimo, dos punteros a nodos del árbol si es un nodo interno.

6. Un nodo interno con q punteros a nodos, $q \leq p$, tiene $q - 1$ valores del campo de indexación.

La estructura de los nodos hoja de un árbol B+ de orden p es la siguiente:

1. Todo nodo hoja es de la forma:

   $((K_1, Pr_1), (K_2, Pr_2), (K_3, Pr_3), \ldots (K_{q-1}, Pr_{q-1}), P_{siguiente})$

   donde $q \leq p$. Cada Pr_i es un puntero al registro de datos que tiene el valor K_i en el campo de indexación, y $P_{siguiente}$ es un puntero al siguiente nodo hoja del árbol.

2. Dentro de cada nodo hoja se cumple: $K_1 < K_2 < \ldots K_{q-1}$, $q \leq p$.

3. Cada nodo hoja tiene, al menos, $\lfloor p/2 \rfloor$ valores.

4. Todos los nodos hoja están al mismo nivel.

Como las entradas en los nodos internos de los árboles B+ contienen valores del campo de indexación y punteros a nodos del árbol, pero no contienen punteros a los registros del fichero de datos, es posible ``empaquetar" más entradas en un nodo interno de un árbol B+ que en un nodo similar de un árbol B. Por tanto, si el tamaño de bloque (nodo) es el mismo, el orden p será mayor para el árbol B+ que para el árbol B. Esto puede reducir el número de niveles del árbol B+, mejorándose así el tiempo de acceso. Como las estructuras de los nodos internos y los nodos hoja de los árboles B+ son diferentes, su orden p puede ser diferente. Se ha demostrado por análisis y simulación que después de un gran número de inserciones y eliminaciones aleatorias en un árbol B, los nodos están ocupados en un 69% cuando se estabiliza el número de valores del árbol. Esto también es verdadero en el caso de los árboles B+. Si

llega a suceder esto, la división y combinación de nodos ocurrirá con muy poca frecuencia, de modo que la inserción y la eliminación se volverán muy eficientes. Cuando los árboles se definen sobre un campo no clave, los punteros a datos pasan a ser punteros a bloques de punteros a datos (se añade un nivel de indirección).

III. Fundamentos de algorítmica

1. Interpretación de algoritmos

- ***Resultado de algoritmos que involucren ejecución secuencial (asignación, condición e iteración)***
- ***Resultado de algoritmos que involucren ejecución recursiva***

2. Aplicación y reconocimiento de algoritmos de búsqueda y ordenamiento interno

- ***Burbuja, de inserción y selección: rangos de n para los cuales son recomendados, estructuras que requieren del lenguaje de implementación, con qué tipos de llaves son convenientes***

Ordenamiento Burbuja (Bubblesort)

1. Descripción. Este es el algoritmo más sencillo probablemente. Ideal para empezar. Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian. ¿Sencillo no?
2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante entera	Tamaño de la lista
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
1. for (i=1; i<TAM; i++)
2.     for j=0 ; j<TAM - 1; j++)
3.         if (lista[j] > lista[j+1])
4.             temp = lista[j];
5.             lista[j] = lista[j+1];
6.             lista[j+1] = temp;
```

3. Un ejemplo. Vamos a ver un ejemplo. Esta es nuestra lista:

4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir, TAM toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - 4 - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3 - 4 - 2 - 5 - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

3 - 4 - 2 - 1 - 5

Repitiendo este proceso vamos obteniendo los siguientes resultados:

3 - 2 - 1 - 4 - 5
2 - 1 - 3 - 4 - 5
1 - 2 - 3 - 4 - 5

4. Optimizando. Se pueden realizar algunos cambios en este algoritmo que pueden mejorar su rendimiento.

• Si observas bien, te darás cuenta que en cada pasada a través de la lista un elemento va quedando en su posición final. Si no te queda claro mira el ejemplo de arriba. En la primera pasada el 5 (elemento mayor) quedó en la última posición, en la segunda el 4 (el segundo mayor elemento) quedó en la penúltima posición. Podemos evitar hacer comparaciones innecesarias si disminuimos el número de éstas en cada pasada. Tan sólo hay que cambiar el ciclo interno de esta manera:

```
for (j=0; j<TAM - i; j++)
```

• Puede ser que los datos queden ordenados antes de completar el ciclo externo. Podemos modificar el algoritmo para que verifique si se han realizado intercambios. Si no se han hecho entonces terminamos con la ejecución, pues eso significa que los datos ya están ordenados. Te dejo como tarea que modifiques el algoritmo para hacer esto :-).
• Otra forma es ir guardando la última posición en que se hizo un intercambio, y en la siguiente pasada sólo comparar hasta antes de esa posición.
5. Análisis del algoritmo. Éste es el análisis para la versión no optimizada del algoritmo:
• Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es estable.
• Requerimientos de Memoria: Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.
• Tiempo de Ejecución: El ciclo interno se ejecuta n veces para una lista de n elementos. El ciclo externo también se ejecuta n veces. Es decir, la complejidad es n * n = O(n2). El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo O(n2).
Ventajas:
• Fácil implementación.
• No requiere memoria adicional.
Desventajas:
• Muy lento.
• Realiza numerosas comparaciones.
• Realiza numerosos intercambios.
Este algoritmo es uno de los más pobres en rendimiento. Si miras la demostración te darás cuenta de ello. No es recomendable usarlo. Tan sólo está aquí para que lo conozcas, y porque su sencillez lo hace bueno para empezar. Ya veremos otros mucho mejores. Ahora te recomiendo que hagas un programa y lo pruebes. Si tienes dudas mira el programa de ejemplo.

Ordenamiento por Inserción

1. Descripción. Este algoritmo también es bastante sencillo. ¿Has jugado cartas?. ¿Cómo las vas ordenando cuando las recibes? Yo lo hago de esta manera: tomo la primera y la coloco en mi mano. Luego tomo la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha, y si es menor a la izquierda (también me fijo en el color, pero omitiré esa parte para concentrarme en la idea principal). Después tomo la tercera y la comparo con las que tengo en la mano, desplazándola hasta que quede en su posición final. Continúo haciendo esto, insertando cada carta en la posición que le corresponde, hasta que las tengo todas en orden. ¿Lo haces así tu también? Bueno, pues si es así entonces comprenderás fácilmente este algoritmo, porque es el mismo concepto.
Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.
2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante Entera	Tamaño de la lista
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
1. for (i=1; i<TAM; i++)
2.     temp = lista[i];
3.     j = i - 1;
4.     while ( (lista[j] > temp) && (j >= 0) )
5.         lista[j+1] = lista[j];
6.         j--;
7.     lista[j+1] = temp;
```

Nota: Observa que en cada iteración del ciclo externo los elementos 0 a i forman una lista ordenada.
3. Un ejemplo. ¿Te acuerdas de nuestra famosa lista?

4 - 3 - 5 - 2 - 1

temp toma el valor del segundo elemento, 3. La primera carta es el 4. Ahora comparamos: 3 es menor que 4. Luego desplazamos el 4 una posición a la derecha y después copiamos el 3 en su lugar.

4 - 4 - 5 - 2 - 1
3 - 4 - 5 - 2 - 1

El siguiente elemento es 5. Comparamos con 4. Es mayor que 4, así que no ocurren intercambios. Continuamos con el 2. Es menor que cinco: desplazamos el 5 una posición a la derecha:

3 - 4 - 5 - 5 - 1

Comparamos con 4: es menor, así que desplazamos el 4 una posición a la derecha:

3 - 4 - 4 - 5 - 1

Comparamos con 3. Desplazamos el 3 una posición a la derecha:

3 - 3 - 4 - 5 - 1

Finalmente copiamos el 2 en su posición final:

2 - 3 - 4 - 5 - 1

El último elemento a ordenar es el 1. Cinco es menor que 1, así que lo desplazamos una posición a la derecha:

2 - 3 - 4 - 5 - 5

Continuando con el procedimiento la lista va quedando así:

2 - 3 - 4 - 4 - 5
2 - 3 - 3 - 4 - 5
2 - 2 - 3 - 4 - 5
1 - 2 - 3 - 4 - 5

Espero que te haya quedado claro.

4. Análisis del algoritmo.

• Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es estable.
• Requerimientos de Memoria: Una variable adicional para realizar los intercambios.
• Tiempo de Ejecución: Para una lista de n elementos el ciclo externo se ejecuta n-1 veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad O(n2).

Ventajas:

• Fácil implementación.
• Requerimientos mínimos de memoria.

Desventajas:

• Lento.
• Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

Ordenamiento por Selección.

1. Descripción. Este algoritmo también es sencillo. Consiste en lo siguiente:

Buscas el elemento más pequeño de la lista.
Lo intercambias con el elemento ubicado en la primera posición de la lista.
Buscas el segundo elemento más pequeño de la lista.
Lo intercambias con el elemento que ocupa la segunda posición en la lista.
Repites este proceso hasta que hayas ordenado toda la lista.

2. Pseudocódigo en C.

Tabla de variables Nombre Tipo Uso
lista Cualquiera Lista a ordenar
TAM Constante entera Tamaño de la lista
i Entero Contador
pos_men Entero Posición del menor elemento de la lista
temp El mismo que los elementos de la lista Para realizar los intercambios

```
1. for (i=0; i<TAM - 1; i++)
2.     pos_men = Menor(lista, TAM, i);
3.     temp = lista[i];
4.     lista[i] = lista [pos_men];
5.     lista [pos_men] = temp;
```

Nota: Menor(lista, TAM, i) es una función que busca el menor elemento entre las posiciones i y TAM-1. La búsqueda es lineal (elemento por elemento). No lo incluyo en el pseudocódigo porque es bastante simple.

3. Un ejemplo. Vamos a ordenar la siguiente lista (la misma del ejemplo anterior :-)):

4 - 3 - 5 - 2 - 1

Comenzamos buscando el elemento menor entre la primera y última posición. Es el 1. Lo intercambiamos con el 4 y la lista queda así:

1 - 3 - 5 - 2 - 4

Ahora buscamos el menor elemento entre la segunda y la última posición. Es el 2. Lo intercambiamos con el elemento en la segunda posición, es decir el 3. La lista queda así:

1 - 2 - 5 - 3 - 4

Buscamos el menor elemento entre la tercera posición (sí, adivinaste :-D) y la última. Es el 3, que intercambiamos con el 5:

1 - 2 - 3 - 5 - 4

El menor elemento entre la cuarta y quinta posición es el 4, que intercambiamos con el 5:

1 - 2 - 3 - 4 - 5

¡Y terminamos! Ya tenemos nuestra lista ordenada. ¿Fue fácil no?

4. Análisis del algoritmo. Estabilidad: Aquí discrepo con un libro de la bibliografía que dice que no es estable. Yo lo veo así: si tengo dos registros con claves iguales, el que ocupe la posición más baja será el primero que sea identificado como menor. Es decir que será el primero en ser desplazado. El segundo registro será el menor en el siguiente ciclo y quedará en la posición adyacente. Por lo tanto se mantendrá el orden relativo. Lo que podría hacerlo inestable sería que el ciclo que busca el elemento menor revisara la lista desde la última posición hacia atrás. ¿Qué opinas tú? Yo digo que es estable, pero para hacerle caso al libro (el autor debe sabe más que yo ¿cierto?:-)) vamos a decir que no es estable. Requerimientos de Memoria: Al igual que el ordenamiento burbuja, este algoritmo sólo necesita una variable adicional para realizar los intercambios. Tiempo de Ejecución: El ciclo externo se ejecuta n veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados. Luego la complejidad es O(n2). Este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad promedio es también O(n2).

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.
- Realiza pocos intercambios.
- Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Este es un algoritmo lento. No obstante, ya que sólo realiza un intercambio en cada ejecución del ciclo externo, puede ser una buena opción para listas con registros grandes y claves pequeñas. Si miras el programa de demostración notarás que es el más rápido en la parte gráfica (por lo menos en un PC lento y con una tarjeta gráfica mala como el mío x-|). La razón es que es mucho más lento dibujar las barras que comparar sus largos (el desplazamiento es más costoso que la comparación), por lo que en este caso especial puede vencer a algoritmos como Quicksort. Bien, ya terminamos con éste. Otra vez te recomiendo que hagas un programa y trates de implementar este algoritmo, de preferencia sin mirar el código ni el pseudocódigo otra vez.

- ***Rápido (quicksort), mezcla (mergesort), ordenamiento de Shell (shellsort), montículo (heapsort): rangos de n para los cuales son recomendados, estructuras que requieren del lenguaje de implementación, son qué tipos de llaves son convenientes***

Ordenamiento Rápido (Quicksort)

1. Descripción. Esta es probablemente la técnica más rápida conocida. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). El algoritmo fundamental es el siguiente: Eliges un elemento de la lista. Puede ser cualquiera (en Optimizando veremos una forma más efectiva). Lo llamaremos elemento de división. Buscas la posición que le corresponde en la lista ordenada (explicado más abajo).
Acomodas los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sublistas (de ahí su nombre). Realizas esto de forma recursiva para cada sublista mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados. Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos contador por la izquierda, y j, al que llamaremos contador por la derecha. El algoritmo es éste: Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento). Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias. Repites esto hasta que se crucen los índices. El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados). Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

2. Pseudocódigo en C.

Tabla de variables		
Nombre	**Tipo**	**Uso**
lista	Cualquiera	Lista a ordenar
inf	Entero	Elemento inferior de la lista
sup	Entero	Elemento superior de la lista
elem_div	El mismo que los elementos de la lista	El elemento divisor
temp	El mismo que los elementos de la lista	Para realizar los intercambios
i	Entero	Contador por la izquierda
j	Entero	Contador por la derecha
cont	Entero	El ciclo continua mientras cont tenga el valor 1

```
Nombre Procedimiento: OrdRap
Parámetros:
  lista a ordenar (lista)
  índice inferior (inf)
  índice superior (sup)
  // Inicialización de variables
  1. elem_div = lista[sup];
  2. i = inf - 1;
  3. j = sup;
  4. cont = 1;
  // Verificamos que no se crucen los límites
  5. if (inf >= sup)
  6.      retornar;
  //  Clasificamos la sublista
  7. while (cont)
  8.      while (lista[++i] < elem_div);
  9.      while (lista[--j] > elem_div);
 10.      if (i < j)
 11.           temp = lista[i];
 12.           lista[i] = lista[j];
 13.           lista[j] = temp;
 14.      else
 15.           cont = 0;
  // Copiamos el elemento de división en su posición final
  16. temp = lista[i];
  17. lista[i] = lista[sup];
  18. lista[sup] = temp;
  // Aplicamos el procedimiento recursivamente a cada sublista
  19. OrdRap (lista, inf, i - 1);
  20. OrdRap (lista, i + 1, sup);
```

Nota:
La primera llamada debería ser con la lista, cero (0) y el tamaño de la lista menos 1 como parámetros.

3. Un ejemplo

Esta vez voy a cambiar de lista ;-D

5 - 3 - 7 - 6 - 2 - 1 - 4

Comenzamos con la lista completa. El elemento divisor será el 4:

5 - 3 - 7 - 6 - 2 - 1 - 4

Comparamos con el 5 por la izquierda y el 1 por la derecha.

5 - 3 - 7 - 6 - 2 - 1 - 4

5 es mayor que cuatro y 1 es menor. Intercambiamos:

1 - 3 - 7 - 6 - 2 - 5 - 4

Avanzamos por la izquierda y la derecha:

1 - 3 - 7 - 6 - 2 - 5 - 4

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

1 - 3 - 7 - 6 - 2 - 5 - 4

7 es mayor que 4 y 2 es menor: intercambiamos.

1 - 3 - 2 - 6 - 7 - 5 - 4

Avanzamos por ambos lados:

1 - 3 - 2 - 6 - 7 - 5 - 4

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):

1 - 3 - 2 - 4 - 7 - 5 - 6

Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:

1 - 3 - 2

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:

1 - 2 - 3

Al llamar recursivamente para cada nueva sublista (lista[0]-lista[0] y lista[2]-lista[2]) se retorna sin hacer cambios (condición 5.).Para resumir te muestro cómo va quedando la lista:

Segunda sublista: lista[4]-lista[6]

7 - 5 - 6

5 - 7 - 6

5 - 6 - 7

Para cada nueva sublista se retorna sin hacer cambios (se cruzan los índices). Finalmente, al retornar de la primera llamada se tiene el arreglo ordenado:

1 - 2 - 3 - 4 - 5 - 6 - 7

Eso es todo. Bastante largo ¿verdad?

4. Optimizando. Sólo voy a mencionar algunas optimizaciones que pueden mejorar bastante el rendimiento de quicksort: Hacer una versión iterativa: Para ello se utiliza una pila en que se van guardando los límites superior e inferior de cada sublista. No clasificar todas las sublistas: Cuando el largo de las sublistas va disminuyendo, el proceso se va encareciendo. Para solucionarlo sólo se clasifican las listas que tengan un largo menor que n. Al terminar la clasificación se llama a otro algoritmo de ordenamiento que termine la labor. El indicado es uno que se comporte bien con listas casi ordenadas, como el ordenamiento por inserción por ejemplo. La elección de n depende de varios factores, pero un valor entre 10 y 25 es adecuado. Elección del elemento de división: Se elige desde un conjunto de tres elementos: lista[inferior], lista[mitad] y lista[superior]. El elemento elegido es el que tenga el valor medio según el criterio de comparación. Esto evita el comportamiento degenerado cuando la lista está prácticamente ordenada.

5. Análisis del algoritmo. Estabilidad: No es estable. Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila. Tiempo de Ejecución: Caso promedio. La complejidad para dividir una lista de n es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:

$f(1) = 1$

$f(n) = n + 2\ f(n/2)$

La forma cerrada de esta expresión es:

$f(n) = n \log 2n$

Es decir, la complejidad es O(n log2n). El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a O(n2). Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

Ventajas:

* Muy rápido
* No requiere memoria adicional.

Desventajas:

* Implementación un poco más complicada.
* Recursividad (utiliza muchos recursos).
* Mucha diferencia entre el peor y el mejor caso.

La mayoría de los problemas de rendimiento se pueden solucionar con las optimizaciones mencionadas arriba (al costo de complicar mucho más la implementación). Este es un algoritmo que puedes utilizar en la vida real. Es muy eficiente. En general será la mejor opción. Intenta programarlo.

- ***Búsqueda: secuencial y binaria***

Existen diferentes tipos de búsqueda, de las cuales mencionaremos tres de las mas importantes:

1.-) Búsqueda secuencial.
2.-) Búsqueda binaria.
3.-) Búsqueda por transformación de claves (Hash).

SECUENCIAL

La más conocida es la búsqueda secuencial y la más utilizada, sin embargo dependerá del método para saber cuál será la más eficiente y la más rápida para poder buscar un elemento deseado. La búsqueda de tipo secuencial se

realiza de una forma ordenada, la cuál se va ir buscando posición por posición el número que se desea buscar. Ejemplo:

pos	0	1	2	3	4
	50	15	56	14	35

Por lo tanto se va ir buscando posición por posición, recorriendo primero la pos. 0, después la pos. 1, así sucesivamente. Si se llega a la pos. 4 (en éste ejemplo) y no se encontró el número buscado, se deduce que el numero no se encuentra. De lo contrario si el número se encuentra en la posición por ejemplo la pos. 2, entonces se detiene hasta ahí la búsqueda (sin continuar hacia las posiciones 3 y 4) y se deduce que el número se ha encontrado.

BINARIA

La búsqueda de tipo binaria se aplica a una serie que ya se encuentra previamente ordenada. Este tipo de búsqueda consiste en posicionar a la mitad de la serie y comparar el número que se desea buscar con el que se tiene al obtener de la mitad de la serie. Aquí se van a tener tres posibles resultados:

a) Si el número a buscar es igual al tomado, entonces ya se encontró el número deseado.

b) Si el número a buscar es menor al tomado en la mitad de la serie, se descarta la mitad derecha y se repite de nuevo el procedimiento sólo con la mitad izquierda.

c) Si el número a buscar es mayor al tomado en la mitad de la serie, se descarta la mitad izquierda y se repite de nuevo el procedimiento sólo con la mitad derecha.

Ejemplo: numero a buscar: 11

pos	0	1	2	3	4	5	6	7	8
	2	7	9	11	20	27	35	49	55

Se toma la mita de la serie, aquí sería la pos. 4 y se compara el número a buscar con el 20, por lo tanto como no son iguales, checamos si el numero es mayor o menor al de la pos. 4, como es menor se descarta la mitad derecha.

pos	0	1	2	3
	2	7	9	11

Se repite el nuevo procedimiento y posiciona a la mitad de la serie (pos 1), y ahora se va a descartar la mitad izquierda por ser mayor al numero en la pos. 1.

pos	2	3
	9	11

Ahora se toma en la posición 2 y como es mayor a 9 se descarta el lado izquierdo y sólo queda el lado derecho. Cuando sólo queda un elemento de la serie y no es igual al número buscado se deduce que no existe, de lo contrario (como es en éste ejemplo) el elemento sí existe

HASH

La búsqueda por transformación de claves es también conocida como técnica Hash. Se tiene que generar una transformación de clave con el fin de obtener la dirección del número donde va a ser colocado. Por ejemplo:

Para un número 31, su transformación de clave por medio de sumas (véase en técnicas de transformación) nos genera una dirección 4, por lo tanto se va a colocar el número 31 en la pos. 4. Ejemplo:

pos	0	1	2	3	4	5	6
					31		

Ahora bien, cuando se tienen dos número que generan la misma dirección se tiene una colisión. Por lo tanto se necesita solucionar la colisión, y tenemos dos tipos de direccionamiento:

1) Direccionamiento Abierto.

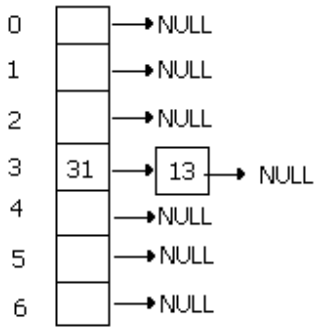
Este tipo de solución de colisiones consiste en colocar en la siguiente posición vacía el número que genero la colisión.

pos	0	1	2	3	4	5	6
					31	41	13

Aquí se tienen que hacer validaciones, como tener un cierto número de posiciones, si al realizar el direccionamiento abierto no se tienen posiciones, números repetidos, etc.

2) Direccionamiento cerrado.

Este tipo de solución de colisiones no se tiene ningún problema con que se repita la misma dirección, utilizando el uso de listas (ver dinámica). Ejemplo:



Las búsquedas de ambos casos requieren primeramente la generación de claves y después realizar la búsqueda. En la abierta se busca desde la dirección hasta una posición vacía y en el cerrado hasta que se encuentra NULL.

3. Aplicación y reconocimiento de ordenamiento externo

- ***Polifase***

El método de mezcla equilíbrada utiliza 2M archivos; esto supone utilizar muchos archivos temporales y es un inconveniente. La mezcla polifásica obtiene mejor partido de cada archivo y se utiliza un número constante de archivos de entrada, Supongamos, por ejemplo, que se dispone de tres archivos, F 1, F2 y F3. F 1 contiene N 1 secuencias, F2 contiene N2 secuencias (N2 < N l). Se realiza la fusión de N2 secuencias de Fl con N2 secuencias de F2, produciendo sobre F3, N2 secuencias.

- ***Intercalación de archivos***

Suponiendo que un archivo desordenado de n registros en memoria externa (disco o cinta) y que se dispone de cuatro áreas contiguas disponibles en la misma. Estas áreas pueden estar en un disco, sobre superficies diferentes de un paquete de discos o en cintas separadas. Dos de ellas son para archivos de entrada; las otras dos son para archivos de salida. Las secuencias se escriben alternativamente sobre archivos de entrada hasta que no quede ningún registro en el archivo original. El segundo paso comienza leyendo las primeras secuencias de los dos archivos de entrada, mezclándolos y escribiendo el más largo sobre uno de los archivos de salida. Las segundas secuencias de los archivos de entrada se mezclan sobre el segundo archivo de salida, etc. Cuando se utilizan cuatro archivos, éste método se conoce como mezcla equilibrada de dos caminos. En general, una mezcla equilibrada de n-caminos utiliza 2*n archivos externos, n cada una para entrada y salida y mezclas de n-caminos. La mezcla equilibrada no requiere de redistribución de registros de salida a n archivos de entrada en cada paso; sin embargo requiere aproximadamente dos veces el número de archivos de salida.

- ***Cascada***

Se disponen las secuencias sobre N archivos FI, F2, ..., FN de tal modo que el número de secuencias sobre FI sea superior al de secuencias sobre F2, etcétera.
numero(F1) > numero(F2) > numero(F3) > ... > numero(FN)
El método prevé un archivo vacío suplementario de numero NI+1.

- ***Mergesort***

El uso de los métodos de ordenación externa es parte importante en el rendimiento global de una computadora y de un sistema de información. Los algoritmos de ordenación externa consideran que los datos se encuentran fuera de la memoria principal. La ordenación interna supone que todos los registros caben completamente en la memoria principal, mientras que en la ordenación externa la mayoría de los datos residirán en un dispositivo de almacenamiento como un disco o una cinta. Los algoritmos de ordenación externa se basan esencialmente en el método conocido como ordenación por mezcla (sort-merge). Estos algoritmos ordenan un archivo con la técnica de particiones (divide y vencerás).

"Mezclas equilibradas"

Suponiendo que un archivo desordenado de n registros en memoria externa (disco o cinta) y que se dispone de cuatro áreas contiguas disponibles en la misma. Estas áreas pueden estar en un disco, sobre superficies diferentes de un paquete de discos o en cintas separadas. Dos de ellas son para archivos de entrada; las otras dos son para archivos de salida. Las secuencias se escriben alternativamente sobre archivos de entrada hasta que no quede ningún registro en el archivo original. El segundo paso comienza leyendo las primeras secuencias de los dos archivos de entrada, mezclándolos y escribiendo el más largo sobre uno de los archivos de salida. Las segundas secuencias de los archivos de entrada se mezclan sobre el segundo archivo de salida, etc. Cuando se utilizan cuatro archivos, éste método se conoce como mezcla equilibrada de dos caminos. En general, una mezcla equilibrada de n-caminos utiliza

2*n archivos externos, n cada una para entrada y salida y mezclas de n-caminos. La mezcla equilibrada no requiere de redistribución de registros de salida a n archivos de entrada en cada paso; sin embargo requiere aproximadamente dos veces el número de archivos de salida.

4. Análisis de algoritmos

- ***"Ordenar" las distintas complejidades en algoritmos y las relaciones de existen entre distintas clases de complejidad***

Diseño del algoritmo: que describe la secuencia ordenada de pasos – sin ambigüedades – que conduzcan a la solución de un problema dado. Expresar el algoritmo como un programa en un lenguaje de programación adecuado.
Ejecución y validación del programa por la computadora.
Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa. Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. Sus características fundamentales son:
a) Debe ser preciso e indicar el orden de realización de cada paso.
b) Debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
c) Debe se finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea debe tener un número finito de pasos.
d) Ha de cumplir tres partes: Entrada – Proceso – Salida.

B. PARADIGMAS DE PROGRAMACIÓN Y LENGUAJES

I. Fundamentos de los lenguajes de programación

1. Elementos de modularización

- ***Subprogramas: diferencia entre procedimientos (subprogramas), funciones y macros.***
- ***Unidades, clases, bibliotecas, paquetes, módulos***

2. Mecanismos de modularización y su rango de aplicación

- ***Vinculación estática y dinámica***
- ***Paso de parámetros: momentos de evaluación de los parámetros, tipos de parámetros, de entrada (*input*, por valor), de salida (*output*, por referencia)***
- ***Valores que regresan las funciones: simples o estructurados, ubicación en el ambiente de ejecución***
- ***Acceso y ocultamiento de la información: datos locales, globales, privados, públicos, distintos niveles de localidad, estructura de bloques, anidamiento lexicográfico, anidamiento dinámico (de ejecución), áreas comunes y registros con campos variantes***
- ***Programación por eventos o interrupciones: diferencias entre interrupciones y excepciones, facilidades para atrapar interrupciones o excepciones***
- ***Coordinación de procesos concurrentes y paralelos: secciones de exclusión mutua, semáforos, mensajes***

3. Paradigmas de programación

- ***Programación estructurada (C)***

CONCEPTO DE PROGRAMACION ESTRUCTURADA

El creciente empleo de los computadores ha conducido a buscar un abaratamiento del desarrollo des software, paralelo a la reducción del costo del hardware obtenido gracias a los avances tecnológicos. Los altos costos del mantenimiento de las aplicaciones en producción normal también han urgido la necesidad de mejorar la productividad del personal de programación.

En la década del sesenta salieron a la luz publica los principios de lo que más tarde se llamo Programación Estructurada, posteriormente se libero el conjunto de las llamadas "Técnicas para mejoramiento de la productividad en programación" (en ingles Improved Programming Technologies, abreviado IPTs), siendo la Programación Estructurada una de ellas.

Los programas computarizados pueden ser escritos con un alto grado de estructuración, lo cual les permite ser mas fácilmente comprensibles en actividades tales como pruebas, mantenimiento y modificación de los mismos. Mediante la programación Estructurada todas las bifurcaciones de control de un programa se encuentran estandarizadas, de forma tal que es posible leer la codificación del mismo desde su inicio hasta su terminación en forma continua, sin tener que saltar de un lugar a otro del programa siguiendo el rastro de la lógica establecida por el programador, como es la situación habitual con codificaciones desarrolladas bajo otras técnicas.

EN programación Estructurada los programadores deben profundizar mas que lo usual al procederá realizar el diseño original del programa, pero el resultado final es más fácil de leer y comprender, el objetivo de u programador profesional al escribir programas de una manera estructurada, es realizarlos utilizando solamente un numero de bifurcaciones de control estandarizados.

EL resultado de aplicar la sistemática y disciplinada manera de elaboración de programas establecida por la Programación Estructurada es una programación de alta precisión como nunca antes había sido lograda. Las pruebas de los programas, desarrollados utilizando este método, se acoplan mas rápidamente y el resultado final con programas que pueden ser leídos, mantenidos y modificados por otros programadores con mucho mayor facilidad.

DEFINICIONES

Programación Estructurada es una técnica en la cual la estructura de un programa, esto es, la interpelación de sus partes realiza tan claramente como es posible mediante el uso de tres estructuras lógicas de control:

a. Secuencia: Sucesión simple de dos o mas operaciones.

b. Selección: bifurcación condicional de una o mas operaciones.

c. Interacción: Repetición de una operación mientras se cumple una condición.

Estos tres tipos de estructuras lógicas de control pueden ser combinados para producir programas que manejen cualquier tarea de procesamiento de información.

Un programa estructurado esta compuesto de segmentos, los cuales puedan estar constituidos por unas pocas instrucciones o por una pagina o más de codificación. Cada segmento tiene solamente una entrada y una salida, estos segmentos, asumiendo que no poseen lazos infinitos y no tienen instrucciones que jamas se ejecuten, se denominan programas propios. Cuando varios programas propios se combinan utilizando las tres estructuras básicas de control mencionadas anteriormente, el resultado es también un programa propio.

La programación Estructurada esta basada en el Teorema de la Estructura, el cual establece que cualquier programa propio (un programa con una entrada y una salida exclusivamente) es equivalente a un programa que contiene solamente las estructuras lógicas mencionadas anteriormente.

Una característica importante en un programa estructurado es que puede ser leído en secuencia, desde el comienzo hasta el final sin perder la continuidad de la tarea que cumple el programa, lo contrario de lo que ocurre con otros estilos de programación. Esto es importante debido a que, es mucho más fácil comprender completamente el trabajo que realiza una función determinada, si todas las instrucciones que influyen en su acción están físicamente contiguas y encerradas por un bloque. La facilidad de lectura, de comienzo a fin, es una consecuencia de utilizar solamente tres estructuras de control y de eliminar la instrucción de desvío de flujo de control, excepto en circunstancias muy especiales tales como la simulación de una estructura lógica de control en un lenguaje de programación que no la posea.

VENTAJAS POTENCIALES

Un programa escrito de acuerdo a estos principios no solamente tendrá una estructura, sino también una excelente presentación.

Un programa escrito de esta forma tiende a ser mucho más fácil de comprender que programas escritos en otros estilos.

La facilidad de comprensión del contenido de un programa puede facilitar el chequeo de la codificación y reducir el tiempo de prueba y depuración de programas. Esto ultimo es cierto parcialmente, debido a que la programación estructurada concentra los errores en uno de los factores más generador de fallas en programación: la lógica.

Un programa que es fácil para leer y el cual esta compuesto de segmentos bien definidos tiende a ser simple, rápido y menos expuesto a mantenimiento. Estos beneficios derivan en parte del hecho que, aunque el programa tenga una extensión significativa, en documentación tiende siempre a estar al día, esto no suele suceder con los métodos convencionales de programación.

La programación estructurada ofrece estos beneficios, pero no se la debe considerar como una panacea ya que el desarrollo de programas es, principalmente, una tarea de dedicación, esfuerzo y creatividad.

TEOREMA DE LA ESTRUCTURA

El teorema de la estructura establece que un programa propio puede ser escrito utilizando solamente las siguientes estructuras lógicas de control: secuencia, selección e iteración.

Un programa de define como propio si cumple con los dos requerimientos siguientes:

a. Tiene exactamente una entrada y una salida para control del programa.

b. Existen caminos seguibles desde la entrada hasta la salida que conducen por cada parte del programa, es decir, no existen lazos infinitos ni instrucciones que no se ejecutan.

Las tres estructuras lógicas de control básicas, se definen de la siguiente forma:

Secuencia: es simplemente la formalización de la idea de que las instrucciones de un programa son ejecutadas en el mismo orden en que ellas aparecen en el programa. En términos de diagrama de flujo la secuencia es representada por una función después de la otra, como se muestra a continuación.

A y B pueden ser instrucciones sencillas hasta módulos completos, lo importante es que sean programas propios, independientemente de su tamaño o complejidad interna. Ay B deben ser programas propios en el sentido en que estos fueron definidos, es decir, que posean solamente una entrada y una salida; la combinación de A seguida por B es también un programa propio, ya que esta unión tiene una entrada y una salida exclusivamente.

Donde la caja externa sugiere que la combinación de A seguida de puede ser tratada como una unidad para propósitos de control.

Selección: Es la escogencia entre dos acciones tomando la decisión en base al resultado de evaluar un predicado. Esta estructura de control es denominada usualmente IFTHENELSE.

Iteración: Esta estructura lógica es utilizada para que se repita la ejecución de un conjunto de instrucciones mientras se cumpla una condición o predicado. Generalmente a esta estructura se le conoce como DOWHILE (hacer mientras).

Se debe comprender claramente que un rectángulo, que representa un modulo en un diagrama, siempre puede ser sustituido por cualquiera de las tres estructuras de control descritas anteriormente; por ejemplo, veamos el diagrama siguiente:

OTRAS ESTRUCTURAS LOGICAS DE CONTROL

Aunque todos los programas pueden ser escritos utilizando solamente las tres estructuras de control descritas anteriormente, es algunas veces, conveniente utilizar algunas estructuras adicionales; a continuación se hará una descripción de esas formas lógicas de control diferentes a las ya definidas:

El Dountil: La estructuras de iteración básica es el DOWHILE, pero existe una estructura que es muy parecida a ella y que aveces es usada, dependiendo del proceso que se este tratando de representar y de las características apropiadas en el lenguaje con el cual se esta trabajando, esta forma de control es la que se llama DOUNTIL.

La diferencia entre el DOWHILE y el DOUNTIL es que en el primero el predicado es probado antes de ejecutar la función, si el predicado es falso la función no es ejecutada; mientras que en el segundo, el predicado es probado

después de ejecutar la función, o sea, que la función siempre será ejecutada al menos una vez, independientemente si el predicado es cierto o falso.

La estructura CASE: Algunas veces resulta de gran ayuda, desde el punto de vista de eficiencias y facilidad de lectura de un programa, tener alguna forma de expresar una desviación del flujo de control hacia múltiples procesos en función del resultado de la evaluación de un predicado; usualmente, a la estructura de control que satisface el requerimiento anterior, se le denomina la estructura *CASE*. Por ejemplo, si es necesario ejecutar una de cien rutinas diferentes en función del valor de un código de 2 dígitos, podemos representar este proceso mediante 100 estructuras *IF*, sin embargo el sentido común nos induce a pensar que no hay razón para adherirnos rígidamente a las tres estructuras básicas de control y en lugar de 100 *IF* usaríamos la estructura *CASE*.

Esta estructura utiliza el valor de una variable para determinar cual, de varias rutinas, será ejecutada.

ETIQUETAS E INSTRUCCIÓN *GOTO:*

Ocasionalmente se habla de la programación estructurada como una técnica de programación que no utiliza GOTO(instrucción de desvío del flujo de control en forma incondicional); si bien es cierto que un programa bien estructurado tiene, o bien ninguna o muy pocas instrucciones GOTO, asumiendo que estamos empleando un lenguaje de programación adecuado, la ausencia de instrucciones GOTO puede ser mal interpretada. Es conveniente que aclaremos este aspecto en este momento.

Un programa bien estructurado gana una parte importante de su fácil comprensión del hecho que puede ser leído en forma secuencial sin desvíos en el flujo de control desde una parte del programa a otra. Esta característica es consecuencia de usar exclusivamente las estructuras lógicas de control estándar (el GOTO no es una de ellas), esta secuencialidad o lectura TOPDOWN es beneficiosa debido a que hay un limite definido para muchos detalles que la mente humana puede abarcar de una vez. Se hace relativamente fácil y rápida la comprensión de la tarea que realiza una instrucción si su función puede ser entendida en términos de unas pocas instrucciones mas, físicamente contiguas y delimitadas.

El problema con la instrucción GOTO es que generalmente aleja al programa realizado de los propósitos descritos y en casos extremos puede hacer que un programa sea esencialmente incomprensible.

No se requieren esfuerzos especiales para eliminar de un programa los GOTO, los cuales han sido, algunas veces, malentendidos como enemigos de la programación estructurada, existen buenas y fundadas razones para no querer usarlos pero no se necesita que se realice un trabajo arduo para eliminarlo; ellos no aparecerán, en general, cuando se utilicen las estructuras lógicas de control, descritas anteriormente. Naturalmente, si escogemos para programar un lenguaje de computación que no posea las estructuras lógicas de control fundamentales, entonces, tendremos que simularlas y seguramente ello implicara el uso de la instrucción GOTO; pero este uso puede hacerse en forma cuidadosamente controlada.

Existen situaciones poco comunes en las cuales el uso de GOTO puede tener ventajas comparado con otras maneras de expresar un proceso; estos casos son excepcionales y usualmente no ocurren en la programación realizada diariamente.

Se deben analizar cuidadosamente las consecuencias de emplear el GOTO, antes de su uso.

SEGMENTACION

Para la comprensión de un programa se haga en forma fácil y rápida es necesario que, al revisarlo, uno no tenga que hojear una gran cantidad de paginas para entender cuales el trabajo que realiza. Una regla practica para lograr estos fines es establecer que cada segmento del programa no exceda, en longitud, a una pagina de codificación, o sea, alrededor de 50 líneas (el significado que se asigna al termino segmento, en este trabajo, no tiene ninguna relación con su significado en relación a las funciones de sistemas operativos o sistemas maneadores de Bases de Datos).

La segmentación no es solamente particionar un programa en trozos cuya longitud sea de unas 50 líneas; esta técnica debe cumplir con ciertas características fundamentales:

a. La segmentación reflejara la división del programa en partes que se relacionen entre sí en forma jerárquica, formando una estructura de árbol. Esta organización puede ser representada gráficamente por un diagrama de procesos, lo que hace más sencillo comprender la relación existente entre un segmento y el resto del programa. Adicionalmente, podemos indicar que, el segmento en la cumbre de la estructura jerárquica

contendrá las funciones de control de mas alto nivel, mientras que los segmentos inferiores en esta organización contendrán funciones detalladas.

b. Una segmentación bien diseñada deberá mostrar, claramente, las relaciones existentes entre las distintas funciones de manera que sea fácil comprender lo que debe hacer el programa y asegurar que efectivamente lo realice. Este hecho, garantizara que los cambios que se efectúen a una parte del programa, durante la programación original o su mantenimiento, no afecten al resto del programa que no ha sufrido cambios.

c. En una segmentación bien realizada la comunicación entre segmentos se lleva a cabo de una manera cuidadosamente controlada. Algunos autores recomiendan que los segmentos consistan en procedimientos y la única comunicación existente entre ellos sea a través de una lista de parámetros, esto reduce la oportunidad de que interactuen entre ellos de una manera indeseada e inentendible.

IDENTACION

El uso de la identacion es importante debido a que, cuando se es consistente en su utilización, facilita la lectura del programa al mostrar en una forma gráfica las relaciones existentes entre las distintas instrucciones.

La identacion puede ser de gran beneficio, tal como se muestra continuación, donde ambos programas realizan la misma función, pero el de la derecha es más fácil de comprender, verificar y corregir.

DIRECTRICES PAR IDENTAR

Debe comprenderse claramente que las líneas siguientes solo pretenden presentar unas directrices de identacion, sin pretender que estas sean las únicas reglas a seguir en este proceso, cada centro de procesamiento deberá establecer sus propias convenciones, sin que sea motivo de preocupación la diferencia respecto a las sugerencias dadas aquí, lo importante es que se establezcan unas normas y se cumplan de manera consistente.

Las siguientes son sugerencias para el desarrollo de una política de identacion en un centro de procesamiento, la idea fundamental es ayudar a que el lector de un programa le sea fácil comprender las relaciones y las funciones existentes en él:

a. En los lenguajes donde se permite el uso de etiquetas, estas deben colocarse lo más externas posibles, por ejemplo comenzando en la columna 2, y deben estar separadas por una línea (siempre que lo permita el lenguaje en uso).

b. Se obtiene consistencia si todas las instrucciones se comienzan en una misma columna, por ejemplo en la columna 4 o cualquier otra ubicada a su derecha.

c. En los lenguajes en que se hagan declaraciones sobre las variables a utilizar, la información quedara mas claramente representada si los atributos declarados se alinean en forma vertical.

d. El uso de lineas en blanco ayuda a mostrar con mas claridad las relaciones existentes entre distintos ítems agrupados en las declaraciones

e. Las instrucciones son mucho mas fáciles de localizar y de cambiar si no se escribe mas de una instrucción por línea.

f. La vision de control de las estructuras lógicas o de los bloques se clarifica si las instrucciones controladas son idénticas por alguna cantidad constante. Se sugiere una identacion de tres espacios.

VENTAJAS DE LA PROGRAMACION ESTRUCTURADA

Con la programación estructurada elaborar programas de computador sigue siendo un albor que demanda esfuerzo, creatividad, habilidad y cuidado. Sin embargo, con este nuevo estilo podemos obtener las siguientes ventajas:

1. - Los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es mas clara puesto que las instrucciones están mas ligadas o relacionadas entre sí, por lo que es más fácil comprender lo que hace cada función.

2. Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas("debugging") se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir mas fácilmente.

3. Reducción de los costos de mantenimiento.

4. Programas más sencillos y más rápidos

5. Aumento de la productividad del programador

6. Se facilita la utilización de las otras técnicas para el mejoramiento de la productividad en programación

LENGUAJE C

Introducción

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje.

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina concreta. Se le suele llamar lenguaje de programación de sistemas debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se puede desarrollar cualquier tipo de aplicación.

La base del C proviene del BCPL, escrito por Martin Richards, y del B escrito por Ken Thompson en 1970 para el primer sistema UNIX en un DEC PDP-7. Estos son lenguajes sin tipos, al contrario que el C que proporciona varios tipos de datos. Los tipos que ofrece son caracteres, números enteros y en coma flotante, de varios tamaños. Además se pueden crear tipos derivados mediante la utilización de punteros, vectores, registros y uniones. El primer compilador de C fue escrito por Dennis Ritchie para un DEC PDP-11 y escribió el propio sistema operativo en C.Introducción al lenguaje C (2).

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan las computadoras. No proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas. No proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Por contra, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de ficheros de encabezamiento, headers, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

Estructura básica de un programa en C

La mejor forma de aprender un lenguaje es programando con él. El programa más sencillo que se puede escribir en C es el siguiente:

```
main( )

{

}
```

Como nos podemos imaginar, este programa no hace nada, pero contiene la parte más importante de cualquier programa C y además, es el más pequeño que se puede escribir y que se compile correctamente. En el se define la función main, que es la que ejecuta el sistema operativo al llamar a un programa C. El nombre de una función C siempre va seguida de paréntesis, tanto si tiene argumentos como si no. La definición de la función está formada por un bloque de sentencias, que esta encerrado entre llaves {}.

Un programa algo más complicado es el siguiente:

```
#include <stdio.h>

main( )

{

printf("Hola amigos!\n");

}
```

Con el visualizamos el mensaje Hola amigos! en el terminal. En la primera línea indica que se tengan en cuenta las funciones y tipos definidos en la librería stdio (standard input/output). Estas definiciones se encuentran en el fichero header stdio.h. Ahora, en la función main se incluye una única sentencia que llama a la función printf. Esta toma como argumento una cadena de caracteres, que se imprimen van encerradas entre dobles comillas " ". El símbolo \n indica un cambio de línea.

Hay un grupo de símbolos, que son tratados como caracteres individuales, que especifican algunos caracteres especiales del código ASCII. Los más importantes son:

\a	Alerta
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical
****	Barra invertida
\'	Comilla simple
\"	Comillas dobles
\OOO	Visualiza un carácter cuyo código ASCII es OOO en octal
\xHHH	Visualiza un carácter cuyo código ASCII es HHH en hexadecimal

Un programa C puede estar formado por diferentes módulos o fuentes. Es conveniente mantener los fuentes de un tamaño no muy grande, para que la compilación sea rápida. También, al dividirse un programa en partes, puede

facilitar la legibilidad del programa y su estructuración. Los diferentes fuentes son compilados de forma separada, únicamente los fuentes que han sido modificados desde la última compilación, y después combinados con las librerías necesarias para formar el programa en su versión ejecutable.

Tipos básicos y variables

Los tipos de datos básicos definidos por C son caracteres, números enteros y números en coma flotante. Los caracteres son representados por char, los enteros por short, int, long y los números en coma flotante por float y double. Los tipos básicos disponibles y su tamaño son:

Char	Carácter	(normalmente 8 bits)
Short	Entero corto con signo	(normalmente 16 bits)
Int	Entero con signo	(depende de la implementación)
Unsigned	Entero sin signo	(depende de la implementación)
Long	Entero largo con signo	(normalmente 32 bits)
Float	Flotante simple	(normalmente 32 bits)
Double	Flotante doble	(normalmente 64 bits)

La palabra unsigned en realidad es un modificador aplicable a tipos enteros, aunque si no se especifica un tipo se supone int. Un modificador es una palabra clave de C que indica que una variable, o función, no se comporta de la forma normal. Hay también un modificador signed, pero como los tipos son por defecto con signo, casi no se utiliza. Las variables son definidas utilizando un identificador de tipo seguido del nombre de la variable. Veamos el siguiente programa:

```
#include <stdio.h>

main()

{

float cels, farh;

farh = 35.0;

cels = 5.0 * ( farh - 32.0 ) / 9.0;

printf("-> %f F son %f C\n", farh, cels );

}
```

En el programa anterior se definen dos variables float, se asigna un valor a la primera y se calcula la segunda mediante una expresión aritmética.

Las asignaciones en C también son una expresión, por lo que se pueden utilizar como parte de otra expresión, pero según que prácticas de este tipo no son muy recomendables ya que reducen la legibilidad del programa. En la instrucción printf, el símbolo %f indica que se imprime un número en coma flotante.

Hay un tipo muy importante que se representa por void que puede significar dos cosas distintas, según su utilización. Puede significar nada, o sea que si una función devuelve un valor de tipo void no devuelve ningún resultado, o puede significar cualquier cosa, como puede ser un puntero a void es un puntero genérico a cualquier tipo de dato. Más adelante veremos su utilización.

Funciones

Un programa C está formado por un conjunto de funciones que al menos contiene la función main. Una función se declara con el nombre de la función precedido del tipo de valor que retorna y una lista de argumentos encerrados entre paréntesis. El cuerpo de la función está formado por un conjunto de declaraciones y de sentencias comprendidas entre llaves. Veamos un ejemplo de utilización de funciones:

```
#include <stdio.h>

#define VALOR 5

#define FACT 120

int fact_i ( int v )

{

int r = 1, i = 0;

while ( i <= v )

{

r = r * i;

i = i + 1;

}

return r;

}

int fact_r ( int v )

{

if ( v == 0 ) return 1;

else return v * fact_r(v-1);

}

main() {

int r, valor = VALOR;

if ( (r = fact_i(valor)) != fact_r(valor) ) printf("Codificación
errónea!!.\n");

else if ( r == FACT ) printf("Codificación correcta.\n");

else printf("Algo falla!!.\n");

}
```

Se definen dos funciones, fact_i y fact_r, además de la función main. Ambas toman como parámetro un valor entero y devuelven otro entero. La primera calcula el factorial de un número de forma iterativa, mientras que la segunda hace lo mismo de forma recursiva.

Todas las líneas que comienzan con el símbolo # indican una directiva del precompilador. Antes de realizar la compilación en C se llama a un precompilador cuya misión es procesar el texto y realizar ciertas sustituciones textuales. Hemos visto que la directiva #include incluye el texto contenido en un fichero en el fuente que estamos compilando. De forma parecida, #define nombre texto sustituye todas las apariciones de nombre por texto. Así, en el fuente, la palabra VALOR se sustituye por el número 5.

El valor que debe devolver una función se indica con la palabra return. La evaluación de la expresión debe dar una valor del mismo tipo de dato que el que se ha definido como resultado. La declaración de una variable puede incluir una inicialización en la misma declaración.

Se debe tener muy en cuenta que en C todos los argumentos son pasados 'por valor'. No existe el concepto de paso de parámetros 'por variable' o 'por referencia'. Veamos un ejemplo:

```
int incr ( int v ) { return v + 1; }

main() {

int a, b;

b = 3;

a = incr(b);

/* a = 4 mientras que b = 3. No ha cambiado después de la llamada. */

}
```

En el ejemplo anterior el valor del parámetro de la función incr, aunque se modifique dentro de la función, no cambia el valor de la variable b de la función main. Todo el texto comprendido entre los caracteres /* y */ son comentarios al programa y son ignorados por el compilador. En un fuente C los comentarios no se pueden anidar.

Expresiones y operadores

Los distintos operadores permiten formar expresiones tanto aritméticas como lógicas. Los operadores aritméticos y lógicos son:

+, -	suma, resta
++, --	incremento, decremento
***, /, %**	multiplicación, división, módulo
>>, <<	rotación de bits a la derecha, izquierda.
&	AND booleano
|	OR booleano
^	EXOR booleano
~	complemento a 1

!	complemento a 2, NOT lógico
==, !=	igualdad, desigualdad
&&, ||	AND, OR lógico
<, <=	menor, menor o igual
>, >=	mayor, mayor o igual

En estos operadores deben tenerse en cuenta la precedencia de operadores y las reglas de asociatividad, que son las normales en la mayoría de lenguajes. Se debe consultar el manual de referencia para obtener una explicación detallada. Además hay toda una serie de operadores aritméticos con asignación, como pueden ser += y ^=.

En la evaluación de expresiones lógicas, los compiladores normalmente utilizan técnicas de evaluación rápida. Para decidir si una expresión lógica es cierta o falsa muchas veces no es necesario evaluarla completamente. Por ejemplo una expresión formada <exp1> || <exp2>, el compilador evalúa primero <exp1> y si es cierta, no evalúa <exp2>. Por ello se deben evitar construcciones en las que se modifiquen valores de datos en la propia expresión, pues su comportamiento puede depender de la implementación del compilador o de la optimización utilizada en una compilación o en otra. Estos son errores que se pueden cometer fácilmente en C ya que una asignación es también una expresión.

Debemos evitar: if ((x++ > 3) || (x < y))

y escribir en su lugar: x++; if ((x > 3) || (x < y))

Hay un tipo especial de expresión en C que se denomina expresión condicional y está representada por los operadores ? : . Su utilización es como sigue: <e> ? <x> : <y>. Se evalúa si e entonces x; si no, y.

```
int mayor ( int a, int b ) {

return ( a > b ) ? TRUE : FALSE;

}

waste_time () {

float a, b = 0.0;

( b > 0.0 ) ? sin(M_PI / 8) : cos(M_PI / 4);

}
```

Conversión de tipos

Cuando escribimos una expresión aritmética a+b, en la cual hay variables o valores de distintos tipos, el compilador realiza determinadas conversiones antes de que evalúe la expresión. Estas conversiones pueden ser para 'aumentar' o 'disminuir' la precisión del tipo al que se convierten los elementos de la expresión. Un ejemplo claro, es la comparación de una variable de tipo int con una variable de tipo double. En este caso, la de tipo int es convertida a double para poder realizar la comparación.

Los tipos pequeños son convertidos de la forma siguiente: un tipo char se convierte a int, con el modificador signed si los caracteres son con signo, o unsigned si los caracteres son sin signo. Un unsigned char es convertido a int con los bits más altos puestos a cero. Un signed char es convertido a int con los bits más altos puestos a uno o cero, dependiendo del valor de la variable.

Para los tipos de mayor tamaño:

- Si un operando es de tipo double, el otro es convertido a double.
- Si un operando es de tipo float, el otro es convertido a float.
- Si un operando es de tipo unsigned long, el otro es convertido a unsigned long.
- Si un operando es de tipo long, el otro es convertido a long.
- Si un operando es de tipo unsigned, el otro es convertido a unsigned.
- Si no, los operandos son de tipo int.

Una variable o expresión de un tipo se puede convertir explícitamente a otro tipo, anteponiéndole el tipo entre paréntesis.

```
void cambio_tipo (void)

{

float a;

int b;

b = 10;

a = 0.5;

if ( a <=(float) b )

menor();

}
```

Control de flujo

Sentencia if

La sentencia de control básica es if (<e>) then <s> else <t>. En ella se evalúa una expresión condicional y si se cumple, se ejecuta la sentencia s; si no, se ejecuta la sentencia t. La segunda parte de la condición, else <t>, es opcional.

```
int cero ( double a )

{

if ( a == 0.0 )

return (TRUE);

else

return (FALSE);

}
```

En el caso que <e> no sea una expresión condicional y sea aritmética, se considera falso si vale 0; y si no, verdadero. Hay casos en los que se deben evaluar múltiples condiciones y únicamente se debe evaluar una de ellas.

Setencia switch

Se puede programar con un grupo de sentencias if then else anidadas, aunque ello puede ser farragoso y de complicada lectura. Para evitarlo nos puede ayudar la sentencia switch. Su utilización es:

```
switch (valor) {

case valor1: <sentencias>

case valor2: <sentencias>

...  default: <sentencias>

}
```

Cuando se encuentra una sentencia case que concuerda con el valor del switch se ejecutan las sentencias que le siguen y todas las demás a partir de ahí, a no ser que se introduzca una sentencia break para salir de la sentencia switch. Por ejemplo:

```
ver_opcion ( char c )

{

switch(c){

case 'a': printf("Op A\n");

break;

case 'b': printf("Op B\n");

break;

case 'c':

case 'd': printf("Op C o D\n");

break;

default: printf("Op ?\n");
```

Setencia while

Otras sentencias de control de flujo son las que nos permiten realizar iteraciones sobre un conjunto de sentencias. En C tenemos tres formas principales de realizar iteraciones. La sentencia while (<e>) <s> es seguramente la más utilizada. La sentencia, o grupo de sentencias <s> se ejecuta mientras la evaluación de la expresión <e> sea verdadera.

```
long raiz ( long valor )

{

long r = 1;

while ( r * r <= valor )

r++;

return r;

}
```

Una variación de la sentencia while es: do <s> while (<e>); En ella la sentencia se ejecuta al menos una vez, antes de que se evalúe la expresión condicional.

Setencia for

Otra sentencia iterativa, que permite inicializar los controles del bucle es la sentencia for (<i>; <e>; <p>) <s>. La sentencia for se puede escribir también como:

<i>;

while (<e>) {

<s>;

<p>;

}

El ejemplo anterior se podría escribir como:

long raiz (long valor)

{

long r;

for (r = 1; r * r <= valor; r++)

;

return r;

}

Break y Continue

Otras sentencias interesantes, aunque menos utilizadas son break y continue. break provoca que se termine la ejecución de una iteración o para salir de la sentencia switch, como ya hemos visto. En cambio, continue provoca que se comience una nueva iteración, evaluándose la expresión de control. Veamos dos ejemplos:

void final_countdown (void)

{

int count = 10;

while (count--> 1)

{

if (count == 4)

start_engines();

if (status() == WARNING)

break;

printf("%d ", count);

}

if (count == 0){

launch();

printf("Shuttle launched\n");

}

```
else
{
printf("WARNING condition received.\n");
printf("Count held at T - %d\n", count );
}
}
d2 ()
{
int f;
for ( f = 1; f <= 50; f++ ) {
if ( f % 2 == 0 )
continue;
printf("%d",f );
}
```

Definición y prototipos de funciones

Los programas sencillos, como los ejemplo planteados hasta ahora, normalmente no necesitan un nivel de estructuración elevado. Pero cuando éstos crecen un poco necesitamos estructurarlos adecuadamente para mantenerlos legibles, facilitar su mantenimiento y para poder reutilizar ciertas porciones de código. El mecanismo C que nos permite esto son las funciones. Con los compiladores, los fabricantes nos proporcionan un conjunto importante de funciones de librería. A veces, nos puede interesar construir nuestras propias librerías. Ya hemos utilizado funciones, pero veamos cómo debemos definirlas.

Los prototipos de funciones son una característica clave de la recomendación ANSI del C. Un prototipo es una declaración que toma la forma:

tipo_resultado nombre_función (tipo_parámetro nombre_parámetro ...);

- int fact_i (int v);
- int mayor (int a, int b);
- int cero (double a);
- long raiz (long valor);
- void final_countdown (void);
- int main (int argc, char **argv);

Observando el prototipo de una función podemos decir exactamente que tipo de parámetros necesita y que resultado devuelve. Si una función tiene como argumento void, quiere decir que no tiene argumentos, al igual que si el resultado es void, no devuelve ningún valor.

En la vieja definición de Kernighan y Ritchie el tipo que devolvía una función se declaraba únicamente si era distinto de int. Similarmente, los parámetros eran declarados en el cuerpo de la función, en lugar de utilizar la lista de parámetros. Por ejemplo:

```
mayor ( a, b )

int a;

int b;

{

...

}
```

Las funciones al viejo estilo se compilan correctamente en muchos compiladores actuales.

Por contra, proporcionan menos información sobre sus parámetros y errores que afecten al tipo de parámetros de llamada a las funciones no pueden ser detectados automáticamente. Por tanto, la declaración de una función debe escribirse igual que su prototipo pero sin el punto y coma final. El cuerpo de la función le sigue encerrado entre llaves.

En un programa que esté formado por distintas partes bien diferenciadas es conveniente utilizar múltiples ficheros fuente. Cada fuente agrupa las funciones semejantes, como por ejemplo en un compilador podríamos tener un fuente para el análisis léxico, otro para el sintáctico y otro para la generación de código. Pero en un fuente necesitaremos funciones que se han definido en otro. Para ello, escribiremos, un fichero de cabecera (header), que contendrá las declaraciones que podemos necesitar en otros fuente. Así, en el fuente que implementa el analizador sintáctico pondremos una línea #include "lexic.h". De esta forma al compilar el módulo sintáctico tendremos todos los prototipos de las funciones del léxico y el compilador podrá detectar malas utilizaciones de las funciones allí definidas.

Construcción de tipos

Los datos del mundo real, normalmente no están formados por variables escalares de tipos los tipos básicos. Por ejemplo, nos puede interesar saber cuántos módulos en C hemos escrito cada semana, a lo largo del año. O también nos interesa tener los datos de cada planeta del Sistema Solar, masa, posición, velocidad y aceleración, para un programa de simulación de la ley de gravitación de Newton. Para resolver el primer caso, C nos permite declarar una variable que sea de tipo vector. Para el segundo, podemos definir un registro para cada elemento.

Un vector es una porción de memoria que es utilizada para almacenar un grupo de elementos del mismo tipo Un vector se declara: tipo nombre [tamaño];. Por ejemplo, int modulo[52];. Aquí 'modulo' es un vector de 52 elementos enteros.

```
main()

{

int f, modulo[52];

for ( f = 0; f< 52; f++ )

modulo[f] = 0;

...

}
```

Cada elemento de un vector es accedido mediante un número de índice y se comporta como una variable del tipo base del vector. Los elementos de un vector son accedidos por índices que van desde 0 hasta N-1 para un vector de N elementos. Los elementos de un vector pueden ser inicializados en la misma declaración:

```
char vocal[5] = {'a', 'e', 'i', 'o', 'u' };

float n_Bode[5] = { 0.4, 0.7, 1, 1.6, 2.8 };
```

También podemos definir vectores multidimensionales. C no impone ninguna limitación al número de dimensiones de un vector. Existe, en cambio, la limitación del tamaño de memoria que podamos utilizar en nuestro ordenador. Por ejemplo, para la declaración de un vector multidimensional podemos escribir:

```
int video[25][80][2];
```

El tamaño de la variable video es proporcional al tamaño del tipo int y al tamaño de cada dimensión. Existe un operador C que nos permite obtener el tamaño de un tipo o de una variable. Este es sizeof() y nos proporciona el tamaño en bytes.

```
if ( sizeof(video) == 80 * 25 * 2 * sizeof(int) )

printf("OK!\n");

else

printf("Algo no funciona.\n");
```

Un tipo vector muy utilizado es la cadena de caracteres (string). Si queremos asignar espacio para un string podemos hacer:

```
char nombre[60], direccion[80];
```

Es un vector C pero con la particularidad de que el propio lenguaje utiliza un carácter especial como marca de final de string. Así en un vector de caracteres de tamaño N podremos almacenar una cadena de N-1 caracteres, cuyo último carácter estará en la posición N-2 y la marca de final de string en la N-1. Veamos un ejemplo:

```
char servei[6] = "SCI";
```

La posición 0 contiene el carácter 'S'; la 1 el 'C'; la 2 el 'l'; la 3 el '\0', marca de final de string. El resto de componentes no están definidas. En la inicialización de strings no se debe indicar el final; ya lo hace el compilador. Para la manipulación de cadenas de caracteres ANSI proporciona el fichero string.h que contiene las declaraciones de un conjunto de funciones proporcionadas con la librería del compilador.

Un registro agrupa distintos tipos de datos en una misma estructura. Los registros son definidos de la forma:

struct nombre{ lista de declaraciones };

Los campos de cada registro pueden ser tipos básicos u otros registros. Por ejemplo:

```
struct planeta {

struct 3D r, v, a;

double masa;

char nom[10];

};

struct 3D {

double x,y,z;

};
```

Los campos de cada registro son accesibles mediante el nombre del registro seguido de punto y el nombre del campo, como por ejemplo venus.r.x = 1.0;. Cada campo se comporta como lo hace su tipo básico. C no proporciona mecanismos de inicialización, ni copia de registros, por lo que debe ser el programador el que los implemente.

A veces los datos se ajustan a series ordenadas en las cuales un elemento sigue, o precede, a otro. Un caso típico son los días de la semana. Si se desea realizar iteraciones con los días de la semana una forma es, por ejemplo, asignar un número a cada día con #define. C proporciona un mecanismo compacto para realizar esto; son las enumeraciones. Una enumeración toma la forma: enum nombre { lista de elementos };. Veamos un ejemplo:

```
void planning ( void )

{

enum diasemana {lunes, martes, miercoles,

jueves, viernes, sabado, domingo };

int dia;

for ( dia = lunes; dia <= viernes; dia++ )

trabajar(dia);

if ( dia == sabado )

salir();

}
```

A cada elemento de la enumeración se le asigna un valor consecutivo, comenzando por 0.

Si se desea que el valor asignado sea distinto se puede hacer de la siguiente forma:

```
enum puntos { t_6_25 = 3, t_zona = 2, t_libre = 1 };
```

Muchas veces es conveniente renombrar tipos de datos para que la escritura del programa se nos haga más sencilla y la lectura también. Esto se puede conseguir con la palabra typedef. Con ella damos un nombre a cierto tipo, o combinación de ellos.

typedef struct planeta PLANETA;

PLANETA mercurio, venus, tierra, marte;

Al igual que podemos inicializar las variables de tipos básicos en la misma declaración, también lo podemos hacer con los registros. Los valores de cada campo de un registro van separados por comas y encerrados entre llaves.

PLANETA mercurio = {{ 0.350, 0, 0 },

{ 0, 0, 0 },

{ 0, 0, 0 },

100,"Mercurio" };

Ambito de funciones y variables.

El ámbito, o visibilidad, de una variable nos indica en que lugares del programa está activa esa variable. Hasta ahora, en los ejemplos que hemos visto, se han utilizado variables definidas en el cuerpo de funciones. Estas variables se crean en la memoria del ordenador cuando se llama a la función y se destruyen cuando la función termina de ejecutarse. Es necesario a veces, que una variable tenga un valor que pueda ser accesible desde todas las funciones de un mismo fuente, e incluso desde otros fuentes.

En C, el ámbito de las variables depende de dónde han sido declaradas y si se les ha aplicado algún modificador. Una variable definida en una función es, por defecto, una variable local. Esto es, que sólo existe y puede ser accedida dentro de la función. Para que una variable sea visible desde una función cualquiera del mismo fuente debe declararse fuera de cualquier función. Esta variable sólo será visible en las funciones definidas después de su declaración. Por esto, el lugar más común para definir las variables globales es antes de la definición de ninguna función. Por defecto, una variable global es visible desde otro fuente. Para definir que existe una variable global que está definida en otro fuente tenemos que anteponer la palabra extern a su declaración. Esta declaración únicamente indica al compilador que se hará referencia a una variable declarada en un módulo distinto al que se compila.

Las variables locales llevan implícito el modificador auto. Este indica que se crean al inicio de la ejecución de la función y se destruyen al final. En un programa sería muy ineficiente en términos de almacenamiento que se crearan todas las variables al inicio de la ejecución. Por contra, en algunos casos es deseable. Esto se consigue anteponiendo el modificador static a una variable local. Si una función necesita una variable que únicamente sea accedida por la misma función y que conserve su valor a través de sucesivas llamadas, es el caso adecuado para que sea declarada local a la función con el modificador static. El modificador static se puede aplicar también a variables globales. Una variable global es por defecto accesible desde cualquier fuente del programa. Si, por cualquier motivo, se desea que una de estas variables no se visible desde otro fuente se le debe aplicar el modificador static. Lo mismo ocurre con las funciones. Las funciones definidas en un fuente son utilizables desde cualquier otro. En este caso conviene incluir los prototipos de las funciones del otro fuente. Si no se desea que alguna función pueda ser llamada desde fuera del fuente en la que está definida se le debe anteponer el modificador static.z

void contar (void)

{

static long cuenta = 0;

cuenta++;

printf("Llamada%ld veces\n", cuenta);

}

Un modificador muy importante es const. Con él se pueden definir variables cuyo valor debe permanecer constante durante toda la ejecución del programa. También se puede utilizar con argumentos de funciones. En esta caso se indica que el argumento en cuestión es un parámetro y su valor no debe ser modificado.

Si al programar la función, modificamos ese parámetro, el compilador nos indicará el error.

```
#define EULER 2.71828

const double pi = 3.14159;

double lcercle (const double r )

{

return 2.0 * pi * r;

}

double EXP ( const double x )

{

return pow (EULER, x );

}

double sinh (const double x )

{

return (exp(x) - exp(-x)) / 2.0;

}
```

Debemos fijarnos que en el ejemplo anterior pi es una variable, la cual no podemos modificar. Por ello pi sólo puede aparecer en un único fuente. Si la definimos en varios, al linkar el programa se nos generará un error por tener una variable duplicada. En el caso en que queramos acceder a ella desde otro fuente, debemos declararla con el modificador extern.

Otro modificador utilizado algunas veces es el register. Este modificador es aplicable únicamente a variables locales e indica al compilador que esta variable debe ser almacenada permanentemente en un registro del procesador del ordenador. Se debe tener en cuenta que de una variable declarada como register no se puede obtener su dirección, ya que está almacenada en un registro y no en memoria.

Punteros

Cada variable de un programa tiene una dirección en la memoria del ordenador. Esta dirección indica la posición del primer byte que la variable ocupa. En el caso de una estructura es la dirección del primer campo. En los ordenadores actuales la dirección de inicio se considera la dirección baja de memoria. Como en cualquier caso las variables son almacenadas ordenadamente y de una forma predecible, es posible acceder a estas y manipularlas mediante otras variables que contenga su dirección. A este tipo de variables se les denomina punteros.

Los punteros C son el tipo más potente y seguramente la otra clave del éxito del lenguaje. La primera ventaja que obtenemos de los punteros es la posibilidad que nos dan de poder tratar con datos de un tamaño arbitrario sin tener que moverlos por la memoria. Esto puede ahorrar un tiempo de computación muy importante en algunos tipos de aplicaciones. También permiten que una función reciba y cambie el valor de una variable. Recordemos que todas las funciones C únicamente aceptan parámetros por valor. Mediante un puntero a una variable podemos modificarla indirectamente desde una función cualquiera.

Un puntero se declara de la forma: tipo *nombre;

```
float *pf;

PLANETA *pp;

char *pc;
```

Para manipular un puntero, como variable que es, se utiliza su nombre; pero para acceder a la variable a la que apunta se le debe preceder de *. A este proceso se le llama indirección. Accedemos indirectamente a una variable. Para trabajar con punteros existe un operador, &, que indica 'dirección de'.

Con él se puede asignar a un puntero la dirección de una variable, o pasar como parámetro a una función.

```
void prueba_puntero ( void ) {

long edad;

long *p;

p = &edad;

edad = 50;

printf("La edad es %ld\n", edad );

*p = *p / 2;

printf("La edad es %ld\n", edad );

}

void imprimir_string ( char string[] ) {

char *p;

for ( p = string; *p != '\0'; p++ )

imprimir_char(*p);

}
```

Definimos un vector de N_PLA componentes de tipo PLANETA. Este tipo está formado por un registro. Vemos que en la función de inicialización del vector el puntero a la primera componente se inicializa con el nombre del vector. Esto es una característica importante de C. La dirección de la primera componente de un vector se puede direccionar con el nombre del vector. Esto es debido a que en la memoria del ordenador, los distintos elementos están ordenados de forma ascendente. Así, SSolar se puede utilizar como &SSolar[0]. A cada iteración llamamos a una función que nos inicializará los datos de cada planeta. A esta función le pasamos como argumento el puntero a la componente en curso para que, utilizando la notación ->, pueda asignar los valores adecuados a cada campo del registro. Debemos fijarnos en el incremento del puntero de control de la iteración, p++. Con los punteros se pueden realizar determinadas operaciones aritméticas aunque, a parte del incremento y decremento, no son muy frecuentes. Cuando incrementamos un puntero el compilador le suma la cantidad necesaria para que apunte al siguiente elemento de la memoria. Debemos fijarnos que esto es aplicable sólo siempre que haya distintas variables o elementos situados consecutivamente en la memoria, como ocurre con los vectores.

De forma similar se pueden utilizar funciones que tengan como parámetros punteros, para cambiar el valor de una variable. Veamos:

```
void intercambio ( void ) {

int a, b;

a = 1;

b = 2;

swap( &a, &b );

printf(" a = %d b = %d\n", a, b );

}

void swap ( int *x, int *y ) {

int tmp;

tmp = *x;

*x = *y;

*y = tmp;
```

La sintaxis de C puede, a veces, provocar confusión. Se debe distinguir lo que es un prototipo de una función de lo que es una declaración de una variable. Así mismo, un puntero a un vector de punteros, etc...

- int f1(); función que devuelve un entero
- int *p1; puntero a entero
- int *f2(); función que devuelve un puntero a entero
- int (*pf)(int); puntero a función que toma y devuelve un entero
- int (*pf2)(int *pi); puntero a función que toma un puntero a entero y devuelve un entero
- int a[3]; vector de tres enteros
- int *ap[3]; vector de tres punteros a entero
- int *(ap[3]); vector de tres punteros a entero
- int (*pa)[3]; puntero a vector de tres enteros
- int (*apf[5])(int *pi); vector de 5 punteros a función que toman un puntero a entero y devuelven un entero.

En los programas que se escriban se debe intentar evitar declaraciones complejas que dificulten la legibilidad del programa. Una forma de conseguirlo es utilizando typedef para redefinir/renombrar tipos.

```
typedef int *intptr;

typedef intptr (*fptr) ( intptr );

fptr f1, f2;
```

Funciones de entrada/salida.

En este apartado y los siguientes vamos a ver algunas de las funciones más importantes que nos proporcionan las librerías definidas por ANSI y su utilización. Como hemos visto hasta ahora, el lenguaje C no proporciona ningún mecanismo de comunicación ni con el usuario ni con el sistema operativo. Ello es realizado a través de las funciones de librería proporcionadas por el compilador.

El fichero de declaraciones que normalmente más se utiliza es el stdio.h. Vamos a ver algunas funciones definidas en él.

Una función que ya hemos utilizado y que, ella y sus variantes, es la más utilizadas para la salida de información es printf. Esta permite dar formato y enviar datos a la salida estándar del sistema operativo.

#include <stdio.h>

int printf (const char *format [, argumentos, ...]);

Acepta un string de formato y cualquier número de argumentos. Estos argumentos se aplican a cada uno de los especificadores de formato contenidos en format. Un especificador de formato toma la forma %[flags][width][.prec][h|l] type. El tipo puede ser:

d, i	entero decimal con signo
o	entero octal sin signo
u	entero decimal sin signo
x	entero hexadecimal sin signo (en minúsculas)
X	entero hexadecimal sin signo (en mayúsculas)
f	coma flotante en la forma [-]dddd.dddd
e	coma flotante en la forma [-]d.dddd e[+/-]ddd
g	coma flotante según el valor
E	como e pero en mayúsculas
G	como g pero en mayúsculas
c	un carácter
s	cadena de caracteres terminada en '\0'
%	imprime el carácter %
p	puntero

Los flags pueden ser los caracteres:

+	siempre se imprime el signo, tanto + como -
-	justifica a la izquierda el resultado, añadiendo espacios al final
blank	si es positivo, imprime un espacio en lugar de un signo +
#	especifica la forma alternativa

En el campo width se especifica la anchura mínima de la forma:

n	se imprimen al menos n caracteres.
0n	se imprimen al menos n caracteres y si la salida es menor, se anteponen ceros
*****	la lista de parámetros proporciona el valor

Hay dos modificadores de tamaño, para los tipos enteros:

l	imprime un entero long
h	imprime un entero short

Otra función similar a printf pero para la entrada de datos es scanf. Esta toma los datos de la entrada estándar del sistema operativo. En este caso, la lista de argumentos debe estar formada por punteros, que indican dónde depositar los valores.

```
#include <stdio.h>

int scanf ( const char *format [, argumentos, ...] );
```

Hay dos funciones que trabajan con strings. La primera lee un string de la entrada estándar y la segunda lo imprime en el dispositivo de salida estándar.

```
#include <stdio.h>

char *gets ( char *s );

int puts ( char *s );
```

También hay funciones de lectura y escritura de caracteres individuales.

```
#include <stdio.h>

int getchar ( void );

int putchar ( int c );
```

Veamos, por ejemplo, un programa que copia la entrada estándar a la salida estándar del sistema operativo, carácter a carácter.

```
#include <stdio.h>

main()

{

int c;

while ( (c = getchar()) != EOF )

putchar(c);

}
```

- ***Programación orientada a objetos (C++ y Java)***

Diferente que la programación estructurada la programación orientada a objetos se enfoca en reducir la cantidad de estructuras de control y reemplazarlo con el concepto de polimorfismo. Aún así los programadores todavía utilizan las estructuras de control (if, while, for, etc...) para implementar sus algoritmos porque en muchos casos es la forma más natural de hacerlo.

1.-CONCEPTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS

1.1 Introducción

Un concepto muy importante introducido por la programación estructurada es la abstracción. La abstracción se puede definir como la capacidad de examinar algo sin preocuparse de los detalles internos. En un programa estructurado es suficiente conocer que un procedimiento dado realiza una tarea específica. El cómo se realiza la tarea no es importante; mientras el procedimiento sea fiable se puede utilizar sin tener que conocer cómo funciona su interior. Esto se conoce como abstracción funcional.
Una debilidad de la programación estructurada aparece cuando programadores diferentes trabajan en una aplicación como un equipo. Dado que programadores diferentes manipulan funciones separadas que pueden referirse a tipos de datos mutuamente compartidos, los cambios de un programador se deben reflejar en el trabajo del resto del equipo. Otro problema de la programación estructurada es que raramente es posible anticipar el diseño de un sistema completo antes de que se implemente realmente.
En esencia, un defecto de la programación estructurada, como se acaba de ver, consiste en la separación conceptual de datos y código. Este defecto se agrava a medida que el tamaño del programa crece.

1.2 Abstracción de datos

La abstracción de datos permite no preocuparse de los detalles no esenciales. Existe en casi todos los lenguajes de programación. Las estructuras de datos y los tipos de datos son un ejemplo de abstracción. Los procedimientos y funciones son otro ejemplo. Sólo recientemente han emergido lenguajes que soportan sus propios tipos abstractos de datos (TAD), como Pascal, Ada, Modula-2 y C++.

1.3 ¿Qué es la programación orientada a objetos?

Se puede definir POO como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción.
Los objetos son en realidad como los tipos abstractos de datos. Un TAD es un tipo definido por el programador junto con un conjunto de operaciones que se pueden realizar sobre ellos. Se denominan abstractos para diferenciarlos de los tipos de datos fundamentales o básicos.
En C se puede definir un tipo abstracto de datos utilizando typedef y struct y la implementación de las operaciones con un conjunto de funciones.
C++ tiene muchas facilidades para definir y utilizar un tipo TAD.
Al igual que los tipos de datos definidos por el usuario, un objeto es una colección de datos, junto con las funciones asociadas, utilizadas para operar sobre esos datos. Sin embargo la potencia real de los objetos reside en las propiedades que soportan: herencia, encapsulación y polimorfismo, junto con los conceptos básicos de objetos, clases, métodos y mensajes.

1.4 Trabajando con objetos

En programación convencional los programas se dividen en dos componentes: procedimientos y datos. Este método permite empaquetar código de programa en procedimientos, pero ¿Qué sucede con los datos? Las estructuras de datos utilizadas en programación son globales o se pasan como parámetros. En esencia los datos se tratan separadamente de los procedimientos.
En POO un programa se divide en componentes que contienen procedimientos y datos. Cada componente se considera un objeto.

Un objeto es una unidad que contiene datos y las funciones que operan sobre esos datos. A los elementos de un objeto se les conoce como miembros; las funciones que operan sobre los datos se denominan métodos (en C++ también se llaman funciones miembro) y los datos se denominan miembros datos. En C++ un programa consta de objetos. Los objetos de un programa se comunican entre sí mediante el paso o envío de mensajes (acciones que debe ejecutar el objeto).
En POO los objetos pueden se cualquier entidad del mundo real:

- Objetos físicos
    * automóviles en una simulación de tráfico
    * aviones en un sistema de control de tráfico aéreo
    * animales mamíferos, etc
- Elementos de interfaces gráficos de usuarios
    * ventanas
    * iconos
    * menús
    * ratones
- Estructuras de datos
    * arrays
    * pilas
    * árboles binarios
- Tipos de datos definidos por el usuario
    * números complejos
    * hora del día

1.5 Definición de objetos

Un objeto es una unidad que contiene datos y las funciones que operan sobre esos datos. Los datos se denominan miembros dato y las funciones métodos o funciones miembro.
Los datos y las funciones se encapsulan en una única entidad. Los datos están ocultos y sólo mediante las funciones miembro es posible acceder a ellos.

1.6 Clases

Una clase es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo. Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. En general, los términos objetos e instancias de una clase se pueden utilizar indistintamente.
Una clase es una colección de objetos similares y un objeto es una instancia de una definición de una clase.
La comunicación con el objeto se realiza a través del paso de mensajes. El envío a una instancia de una clase produce la ejecución de un método o función miembro. El paso de mensajes es el término utilizado para referirnos a la invocación o llamada de una función miembro de un objeto.

1.7 Mensajes: activación de objetos

Los objetos pueden ser activados mediante la recepción de mensajes. Un mensaje es simplemente una petición para que un objeto se comporte de una determinada manera, ejecutando una de sus funciones miembro. La técnica de enviar mensajes se conoce como paso de mensajes.

Estructuralmente un mensaje consta de tres partes:
- la identidad del objeto receptor
- la función miembro del receptor cuya ejecución se ha solicitado
- cualquier otra información adicional que el receptor pueda necesitar para jecutar el método requerido.

En C++, la notación utilizada es nombre_del_objeto.función_miembro

Ejemplo: Se tiene un objeto o1 con los siguientes miembros dato: nombre_alumno y curso y con las funciones miembro: leer_nombre e imprimir. Si el objeto o1 recibe el mensaje imprimir, esto se expresa:

```
o1.imprimir()
```

La sentencia anterior se lee: "enviar mensaje imprimir al objeto o1". El objeto o1 reacciona al mensaje ejecutando la función miembro de igual nombre que el mensaje.

El mensaje puede llevar parámetros:

```
o1.leer_nombre("Pedro Pérez")
```

Sin los mensajes los objetos que se definan no podrán comunicarse con otros objetos. Desde un punto de vista convencional, el paso de mensajes no es más que el sinónimo de llamada a una función.

1.8 Programa orientado a objetos

Un programa orientado a objetos es una colección de clases. Necesitará una función principal que cree objetos y comience la ejecución mediante la invocación de sus funciones miembro.
Esta organización conduce a separar partes diferentes de una aplicación en distintos archivos. La idea consiste en poner la descripción de la clase para cada una de ellas en un archivo separado. La función principal también se pone en un archivo independiente. El compilador ensamblará el programa completo a partir de los archivos independientes en una única unidad.
En realidad, cuando se ejecuta un programa orientado a objetos, ocurren tres acciones:

1. Se crean los objetos cuando se necesitan
2. Los mensajes se envían desde uno objetos y se reciben en otros
3. Se borran los objetos cuando ya no son necesarios y se recupera la memoria ocupada por ellos

1.9 Herencia

La herencia es la propiedad que permite a los objetos construirse a partir de otros objetos.
Una clase se puede dividir en subclases. En C++ la clase original se denomina clase base; las clases que se definen a partir de la clase base, compartiendo sus características y añadiendo otras nuevas, se denominan clases derivadas.
Las clases derivadas pueden heredar código y datos de su clase base añadiendo su propio código y datos a la misma.
La herencia impone una relación jerárquica entre clases en la cual una clase hija hereda de su clase padre. Si una clase sólo puede recibir características de otra clase base, la herencia se denomina herencia simple.
Si una clase recibe propiedades de más de una clase base, la herencia se denomina herencia múltiple.

1.10 Polimorfismo

En un sentido literal, significa la cualidad de tener más de una forma. En el contexto de POO, el polimorfismo se refiere al hecho de que una misma operación puede tener diferente comportamiento en diferentes objetos. Por ejemplo, consideremos la operación sumar. El operador + realiza la suma de dos números de diferente tipo. Además se puede definir la operación de sumar dos cadenas mediante el operador suma.

2.- C++: UN C “MEJOR”

2.1 Comentarios

C++ soporta dos tipos de comentarios:
1. Viejo estilo C: Entre /* y */.
2. Nuevo estilo C++: Comienzan con //. Su efecto termina cuando se alcanza el final de la línea actual.

2.2 Identificadores

Son los nombres elegidos para las variables, constantes, funciones, clases y similares. El primer carácter debe ser una letra o un subrayado. El resto del nombre puede contener dígitos. Los identificadores que comienzan con dos subrayados están reservados para uso interno del compilador C++.

2.3 Constantes

C++ permite utilizar varios tipos de constantes:
1. Constantes enteras 44 0 -345
2. Constantes enteras muy grandes. Se identifican situando una L al final de la constante entera 33L -105L
3. Constantes octales o hexadecimales. Un 0 a la izquierda indica una constante octal y un 0x o bien 0X indican una constante hexadecimal
   0 02 077 0123 equivalen a 0 2 63 83 en octal
   0x0 0x2 0x3F 0x53 equivalen a 0 2 63 83 en hexadecimal
4. Constantes reales (coma flotante) 0.0 3.1416 -99.2
C++ permite especificar constante de coma flotante de simple precisión (sufijo f o F) y doble precisión larga (sufijo l o L). 32.0f 3.1416L
5. Constantes carácter 'z' '5'
6. Constantes cadena "hola" "hoy es lunes"

2.4 Tipos de datos

C++, igual que C, contiene tipos fundamentales y tipos derivados o estructurados.
Los fundamentales son: int, char, long int, float, double, long double.

- Tipo vacío. El tipo vacío (void) se utiliza principalmente para especificar:
    * Funciones que no devuelven valores.
    * Punteros void, que referencian a objetos cuyo tipo es desconocido.

- Tipos enumerados. Un tipo enumerado o enumeración está construido por una serie de constantes simbólicas enteras. Los tipos enumerados se tratan de modo ligeramente diferente en C++ que en ANSI C. El nombre de la etiqueta enum se considera como un nombre de tipo igual que las etiquetas de struct y union. Por tanto se puede declarar una variable de enumeración, estructura o union sin utilizar las palabras enum, strcut o union.
C define el tipo de enum de tipo int. En C++, sin embargo, cada tipo enumerado es su propio tipo independiente. Esto significa que C++ no permite que un valor int se convierta automáticamente a un valor enum. Sin embargo, un valor enumerado se puede utilizar en lugar de un int.

Ej.
```
enum lugar{primero,segundo,tercero};
lugar pepe=primero; //correcto
int vencedor=pepe;  //correcto
lugar juan=1;       //incorrecto
```
La última sentencia de asignación es aceptable en C pero no en C++, ya que 1 no es un valor definido en lugar.

- Tipos referencia. Las referencias son como alias. Son alternativas al nombre de un objeto. Se define un tipo referencia haciéndole preceder por el operador de dirección &. Un objeto referencia, igual que una constante debe ser inicializado.

```
int a=50;
int &refa=a;  //correcto
int &ref2a;   //incorrecto: no inicializado
```
Todas las operaciones efectuadas sobre la referencia se realizan sobre el propio objeto:
```
refa+=5; equivale a sumar 5 a a, que vale ahora 55
int *p=&refa; inicializa p con la dirección de a
```

2.5 Operadores especiales de C++

:: Resolución de ámbito (o alcance)
.* Indirección (eliminación de referencia directa) un puntero a un miembro de una clase
->* Indirección (eliminación de referencia directa) un puntero a un miembro de una clase
new Asigna (inicializa) almacenamiento dinámico
delete Libera almacenamiento asignado por new

2.6 Declaraciones y definiciones

Los términos declaración y definición tienen un significado distinto aunque con frecuencia se intercambian.
Las declaraciones se utilizan para introducir un nombre nombre al compilador, pero no se asigna memoria. Las definiciones asignan memoria.
En C++ cuando se declara una estructura se proporciona su nombre al compilador pero no se asigna memoria. Cuando se crea una instancia de la estructura es cuando se asigna memoria.
En C todas las declaraciones dentro de un programa o función deben hacerse al principio del programa o función; en otras palabras las declaraciones dentro de un ámbito dado deben ocurrir al principio de ese ámbito.. Todas las declaraciones globales deben aparecer antes de cualquier función y cualquier declaración local debe hacerse antes de cualquier sentencia ejecutable.
C++, por el contrario, permite mezclar datos con funciones y código ejecutable: trata una declaración como un tipo de sentencia y permite situarla en cualquier parte como tal. Esta característica de C++ es muy cómoda ya que permite declarar una variable cuando se necesite e inicializarla inmediatamente.
El ámbito de una variable es el bloque actual y todos los bloques subordinados a él. Su ámbito comienza donde aparece la declaración. Las sentencias C++ que aparecen antes de la declaración no pueden referirse a la variable incluso aunque aparezcan en el mismo bloque que la declaración de la variable.

2.7 Moldes (cast)

C++ soporta dos formas diferentes de conversiones forzosas de tipo explícitas:

```
 int f=0;
long l= (long) f; molde tradicional, tipo C
long n = long (f);  molde nuevo, tipo C++
```

La sintaxis nombre_tipo (expresión) se conoce como notación funcional y es preferible por su legibilidad.

2.8 El especificador constante (const)

Una constante es una entidad cuyo valor no se puede modificar, y en C++, la palabra reservada const se utiliza para declarar una constante.

```
const int longitud = 20;
char array[longitud];  // válido en C++ pero no en C
```

Una vez que una constante se declara no se puede modificar dentro del programa.
En C++ una constante debe ser inicializada cuando se declara, mientras que en ANSI C, una constante no inicializada se pone por defecto a 0.
La diferencia más importante entre las constantes en C++ y C es el modo como se declaran fuera de las funciones. En ANSI C se tratan como constantes globales y se pueden ver por cualquier archivo que es parte del programa. En C++, las constantes que se declaran fuera de una función tienen ámbito de archivo por defecto y no se pueden ver fuera del archivo en que están declaradas. Si se desea que una constante se vea en más de un archivo, se debe declarar como extern.
En C++ las constantes se pueden utilizar para sustituir a #define.

```
En C
 En C++

#define PI 3.141592    const PI = 3.141592
#define long 128          const long = 128
```

2.9 Punteros y direcciones de constantes simbólicas

No se puede asignar la dirección de una constante no simbólica a un puntero no constante, pero sí se puede desreferenciar el puntero y cambiar el valor de la constante.

```
const int x=6;
int *ip;
ip = &x;    //error
*ip=7;
```

Punteros a un tipo de dato constante
Se puede declarar un puntero a un tipo de dato constante pero no se puede desreferenciar.

```
const int x=6;
const int *ip;   //puntero a una constante entera
int z;
ip=&x;
ip=&z;
*ip=10;     //error: no se puede desreferenciar este tipo de puntero y modificar z
z=7;         //válido: z no es una constante y se puede cambiar
```

El puntero a un tipo constante es muy útil cuando se desea pasar una variable puntero como un argumento a una función pero no se desea que la función cambie el valor de la variable a la que está apuntada.

```
struct ejemplo
{
        int x;
        int y;
};
void funcion(const ejemplo *);
void main()
{
  ejemplo e={4,5};
  funcion(&e);   //convertirá un puntero no constante a un puntero const
```

```
}
void funcion(const ejemplo *pe)
{   ejemplo i;
    i.x=pe->x;
    i.y=pe->y;
    pe->x=10;   }   // error: no se puede desreferenciar pe
```

Punteros constantes
Se puede definir una constante puntero en C++. Las constantes puntero se deben inicializar en el momento en que se declaran. Se leen de derecha a izquierda.

```
void main()
{
  int x=1,y;
  int *const xp= &x;  // xp es un puntero constante a un int
  xp=&y;     // error: el puntero es una constante y no se puede asignar un nuevo
                 valor a punteros constantes
  *xp=4;     }
```

2.10 El especificador de tipo volatile

La palabra reservada volatile se utiliza sintácticamente igual que const, aunque en cierto sentido tiene sentido opuesto.
La declaración de una variable volátil indica al compilador que el valor de esa variable puede cambiar en cualquier momento por sucesos externos al control del programa. En principio, es posible que las variables volátiles se puedan modificar no sólo por el programador sino también por hardware o software del sistema (rutina de interrupción).

```
                                volatile int puerto;
```

2.11 Sizeof (char)

En C, todas las constantes char se almacenan como enteros. Esto significa que en C un carácter ocupa lo mismo que un entero. En C++ un char se trata como un byte, y no como el tamaño de un int. Por ejemplo en una máquina con una palabra de 4 bytes sizeof('a') se evalúa a 1 en C++ y a 4 en C.

2.12 Punteros a void

En C, una variable puntero siempre apunta a un tipo de dato específico, de modo que el tipo de dato es importante para aritmética de punteros. Si se intenta inicializar una variable puntero con el tipo de dato incorrecto, el compilador genera un mensaje de error. En C se pueden moldear los datos al tipo de dato correcto:

```
void main()
{
  int a=1,*p;
  double x=2.4;
  p=&a;   //válido: puntero y variable son del mismo tipo
  p=&x;   //error
  p=(int *)&x; //válido
}
```

En C++ se puede crear un puntero genérico que puede recibir la dirección de cualquier tipo de dato.

```
void main()
{   void *p;
    int a=1;
    double x=2.4;
    p=&a;
    p=&x;   }
```

No se puede desrefernciar un puntero void.

```
void main()
{  void *p;
   double x=2.5;
   p=&x;
   *p=3.6;    // error: se desreferencia a un puntero void
```

```
}
```

2.11 Salidas y entradas

Las operaciones de salida y entrada se realizan en C++, al igual que en C, mediante flujos (streams) o secuencias de datos. Los flujos estandar son cout (flujo de salida) y cin (flujo de entrada). La salida fluye normalmente a la pantalla y la entrada representa los datos que proceden de teclado. Ambos se pueden redireccionar.

1. Salida
El flujo de salida se representa por el identificador cout, que es en realidad un objeto. El operador << se denomina operador de inserción y dirige el contenido de la variable situada a su derecha al objeto situado a su izquierda. El equivalente en C de cout es printf.
El archivo de cabecera iostream.h contiene las facilidades standard de entrada y salida de C++.
En C++, los dispositivos de salida estandar no requieren la cadena de formato.
Se pueden utilizar también diferente tipos de datos, enviando cada uno de ellos a la vez al flujo de salida. El flujo cout discierne el formato del tipo de dato, ya que el compilador C++ lo descifra en el momento de la compilación.
El operador de inserción se puede utilizar repetidamente junto con cout.

```
include <iostream.h>
void main()
{
  int a=4;
  float b=3.4;
  char *texto="hola\n";
  cout<< "entero " << a << " real " << b << " mensaje " << texto;
}
```

Salida con formato
C++ asocia un conjunto de manipuladores con el flujo de salida, que modifican el formato por defecto de argumentos enteros. Por ejemplo, valores simbólicos de manipuladores son dec, oct y hex que visualizan representaciones decimales, octales y hexadecimales de variable

2. Entrada
C++ permite la entrada de datos a través del flujo de entrada cin. El objeto cin es un objeto predefinido que corresponde al flujo de entrada estandar. Este flujo representa los datos que proceden del teclado. El operador >> se denomina de extracción o de lectura de. Toma el valor del objeto flujo de su izquierda y lo situa en la variable situada a su derecha.

2.12 El operador de resolución de ámbito ::

C es un lenguaje estructurado por bloques. C++ hereda la misma noción de bloque y ámbito. En ambos lenguajes, el mismo identificador se puede usar para referenciar a objetos diferentes. Un uso en un bloque interno oculta el uso externo del mismo nombre. C++ introduce el operador de resolución de ámbito o de alcance.
El operador :: se utiliza para acceder a un elemento oculto en el ámbito actual. Su sintaxis es :: variable

Ejemplo:

```
#include <iostream.h>
int a;
void main()
{
  float a;
  a=1.5;
  ::a=2;
  cout << "a local " << a << "\n";
  cout << "a global " << ::a << "\n";
}
```

Este programa visualiza: a local 1.5
a global 2
Este operador se utilizará también en la gestión de clases.

2.13 Estructuras y uniones

Los tipos definidos por el usuario se definen como estructuras, uniones, enumeraciones o clases. Las estructuras y uniones son tipos de clase (clase, es el tipo de dato fundamental en la programación orientada a objetos).
El tipo struct permite agregar componentes de diferentes tipos y con un solo nombre. Las estructuras en C++ se declaran como en C:

```
struct datos
   {
     int num;
     char nombre[20];
   };
```

El nuevo tipo de dato definido puede tener instancias que se declaran:

```
datos persona;              //declaración C++
struct datos persona;       // declaración C
```

En C++ debe evitarse el uso de typedef.
El mismo convenio aplicado a las estructuras, se aplica a las uniones.
Un tipo especial de unión se ha incorporado a C++ :unión anónima. Una unión anónima declara simplemente un conjunto de elementos que comparten la misma dirección de memoria; no tiene nombre identificador y se puede acceder directamente a los elementos por su nombre.

Un ejemplo:

```
union
  {
    int i;
    float f;
  };
```

Tanto i como f comparten la misma posición de memoria. A los miembros de esta unión se puede acceder directamente en el ámbito en que está declarada. Por tanto, en el ejemplo, la sentencia i=3 sería aceptable.
Las uniones anónimas son interesantes en el caso en que se defina una unión en el interior de una estructura:

```
struct registro
    {
        union
              {
                int num;
                float salario;
              };
        char *telefono;
    };
 registro empleado;
```

Para acceder al nombre de un campo dentro de esta estructura:

```
empleado.sueldo;
```

2.14 Asignación dinámica de memoria: new y delete

En C la asignación dinámica de memoria se manipula con las funciones malloc() y free(). En C++ se define un método de hacer asignación dinámica utilizando los operadores new y delete.
En C:

```
void main()
{
  int *z;
  z= (int*) malloc(sizeof(int));
  *z=342;
  printf("%d\n",*z);
  free(z);
}
```

En C++:

```
void main()
{
  int *z=new int;
```

```
  *z=342;
  cout << z;
  delete z;
}
```

El operador new está disponible directamente en C++, de modo que no se necesita utilizar ningún archivo de cabecera; new se puede utilizar con dos formatos:

```
new tipo                  // asigna un único elemento
new tipo[num_eltos]       // signa un array
```

Si la cantidad de memoria solicitada no está disponible, el operador new proporciona el valor 0.
El operador delete libera la memoria signada con new.

```
delete variable
delete [n] variable
```

new es superior a malloc por tres razones:
1. new conoce cuánta memoria se asigna a cada tipo de variable.
2. malloc() debe indicar cuánta memoria asignar.
3. new hace que se llame a un constructor para el objeto asignado y malloc no puede.

delete produce una llamada al destructor en este orden:
1. Se llama al destructor
2. Se libera memoria

delete es más seguro que free() ya que protege de intentar liberar memoria apuntada por un puntero nulo.

- ***Programación funcional (Lisp)***

LISP (Procesamiento de Listas)

El LISP es, como se sabe, uno de los lenguajes de programación en uso más antiguos. A finales de los años 50, John McCarthy, a quien se considera como el padre de la Inteligencia Artificial, diseñó este lenguaje para servir como herramienta de programación en esta disciplina. La idea de LISP surgió a partir de un sistema lógico llamado "lambda calculus" desarrollado por Alonzo Church. Existen diversas variantes (o dialectos) de LISP, entre las cuales se encuentran Scheme, T, etc. LISP llegó a ser fundamental como lenguaje de programación para las investigaciones de Inteligencia Artificial, y sigue aún hoy siendo uno de los más utilizados en este campo. En la década de los '80 se intentó estandarizar el lenguaje. Como resultado surgió el Common LISP cyas especificaciones se recogen en Common LISP: The Language, 2nd Edition (CLTL2). Common LISP es actualmente el dialecto más difundido y la base para el desarrollo de numerosas implementaciones. Una gran ventaja de usar LISP es que suele citarse frente a las demás alternativas de los lenguajes usados para la Inteligencia Articial es su gran flexibilidad. El hecho de que tanto los programas como los datos compartan una misma estructura, las listas, hace que los propios programas se puedan modificar a si mismos. Para ser mas precisos, en LISP se pueden crear con gran facilidad lenguajes o estructuras de datos adaptados al tipo de problema del que se trate. Las armas fundamentales para esta flexibilidad son los macros y la capacidad para crear funciones en tiempo de ejecución. LISP constituye un entorno en el que se pueden crear prototipos (y posteriormente productos finales) con una gran rapidez y comodidad; suponiendo que el programador está familiarizado con el lenguaje, por supuesto. Esta característica se debe a que el LISP en cierto sentido proporciona un entorno de muy alto nivel (en términos de programación) ya que, el Common LISP, proporciona un conjunto de más de 700 funciones primitivas lo que permite despreocuparse de numerosos detalles de implementación de bajo nivel a diferencia de lo que ocurre con lenguajes como el C (salvo que se disponga de una buena librería para desarrollar este tipo de aplicaciones). Además, el carácter interpretado y también compilado del LISP hace que el desarrollo del programa sea muy cómodo sobre todo teniendo en cuenta las posibilidades de sus herramientas de depuración. Una de las características de LISP es la posibilidad de tratar las propias funciones como datos. En LISP, funciones e incluso programas enteros pueden ser utilizados directamente como entrada a otros programas o subrutinas. En esto el prototipo para la concepción del lenguaje ha sido la estructura de las funciones matemáticas. Todos sabemos cómo resolver una expresión del tipo (8 * ((17 + 3) / 4)). Primero hallaríamos el resultado de 17 + 3, que entonces dividiríamos entre 4, para el resultado multiplicarlo por 8. Es decir, que iríamos resolviendo los paréntesis más interiores y pasando los resultados a las operaciones descritas en los paréntesis que los contienen.
(* 8 (/ (+ 3 17) 4)) sería la función LISP equivalente.

*, / y + son nombres de funciones LISP. Los números en (+ 3 17) son los argumentos que se pasan a la función '+'. Pero en (/ (+ 3 17) 4) a la función '/' se le está pasando un argumento numérico 4, pero también (+ 3 17), otra función con dos argumentos numéricos. El lenguaje Lisp es el segundo lenguaje de programación más antiguo de los actualmente en uso (fortran es el mas viejo) fue creado a fines de los 50´s por John McCarty. Esta especialmente concebido para el tratamiento de lenguaje simbólico. Sus elementos estructurales más importantes son las listas y los símbolos. Lisp es el lenguaje preferido e indispensable para todo aquel que trabaja en Inteligencia Artificial. Las herramientas de desarrollo de sistemas expertos más potentes están escritas en lisp y lo utilizan como lenguaje de programación. Lisp es un lenguaje en el cual no existen diferencias entre la representación de las estructuras de datos y las de los programas: ambas se estructuran mediante listas. Esto permite tratar los datos como programas y viceversa. Por su definición resulta optimo para programar funciones recursivas.
Caracteres detallados más comunes de la sintaxis de LISP:
(paréntesis izquierdo indica el comienzo de una lista de objetos. La lista puede contener un numero indefinido de objetos e incluso ninguno o estar anidadas. (átomo (átomo) (átomo)) es una lista con tres objetos.
) paréntesis derecho termina una lista de objetos.
´ **apóstrofo**, también llamado "quote" seguido de un objeto, es la abreviación de la función, así pues ´hola es equivalente a (quote hola).
, La coma se emplea como un blanco, aunque su uso resulta anacrónico.
; punto y coma indica el principio de un comentario; apartir de su aparición, el sistema ignora todos los objetos siguientes hasta el fin de la línea.
" Dobles comillas sirven para delimitar las de caracteres o cadenas.
En UPCLISP son caracteres especiales:
! Signo de admiración indica el principio de un comentario.
] Corchete derecho es un superparentesis que sirve para balancear (cerrar) todos los paréntesis izquierdos que no tengan compañero hasta su aparición. Ejemplo: (((( )] es equivalente a (((( )))).
El formato de las funciones es: (nombre-funcion Arg1 Arg2)
Nombre y descripción de su funcionamiento, tipo de argumentos que recibe y los errores que devuelve. Lisp no hace distinciones entre mayúsculas y minúsculas a excepción de la versión commonlisp. Las listas son el tipo de datos más importante en lisp. Una lista es un conjunto de elementos que están encerrados entre paréntesis, estos elementos pueden ser números, símbolos, otras listas, etc. Cada elemento de una lista esta separado del siguiente mediante un espacio en blanco. Una lista esta formada por celdas cons.. Una celda cons es una función que crea y devuelve una nueva celda cons..Por tanto, sirve para construir listas (cons. Primer-elemento resto-de-la-lista).
Cuando el segundo elemento no es una lista se obtiene un par con punto(dotted pair).ejemplo:

- (cons ´a nil)
- (a)
- (cons ´a ´(b c))
- (a b c)
- (cons ´a ´b)
- (a.b)

Una celda cons a su vez esta formada por una celda car y cdr. La celda car contiene el primer elemento de la lista mientras que la celda cdr el resto de la lista. Así car de la lista (a b c d) es el elemento a , mientras que el cdr de esta lista es la lista (b c d). Una lista que no contiene elementos se puede representar de dos maneras: ( ) o bien NIL. El car y el cdr de nil (lista vacía) es NIL. El último elemento apunta a nil. Existe un tipo especial de lista con punto (dotted list) .este tipo de lista se caracteriza porque las celdas cdr de su ultimo elemento no apunta a nil sino a otro elemento. Ejemplo:
(a b c . d) esta lista tiene tres elementos y no cuatro. Su último elemento es c.d.
el car de su ultimo elemento es c, mientras que su cdr no es NIL sino d.

- ***Programación lógica (Prolog)***

PROgramming in LOGic (PROLOG), es otro de los lenguajes de programación ampliamente utilizados en IA. PROLOG fue desarrollado en Francia, en 1973 por Alain Colmenauer y su equipo de investigación en la Universidad de Marseilles. Inicialmente fue utilizado para el procesamiento de lenguaje natural, pero posteriormente se popularizó entre los desarrolladores de aplicaciones de IA por su capacidad de manipulación simbólica. Utilizando los resultados del grupo francés, Robert Kowalski de la Universidad de Edimburgo, en Escocia, desarrolló la teoría de la programación lógica. La sintaxis propuesta por Edimburgo, se considera el estándar de facto del PROLOG.

A partir de 1981 tuvo una importante difusión en todo el mundo, especialmente porque los japoneses decidieron utilizar PROLOG para el desarrollo de sus sistemas de computación de quinta generación. Actualmente existen varios dialectos del PROLOG para diferentes plataformas.

- ***Conceptos de programación visual***

La primera metodología para la programación de aplicaciones fue la utilización directa de llamadas a las funciones del API (Application Programming Interface) de Windows. Este mecanismo suponía una pesada tarea de codificación para crear aplicaciones con simples elementos tales como menús, ventanas, cuadros de diálogo, etc. El entorno de desarrollo por excelencia era un compilador de C, con lenguaje de tercera generación (3GL).

A medida que el entorno Windows fue extendiéndose y ofrecía nuevas posibilidades tales como DDE y OLE, era necesario un conjunto de herramientas que permitiese un desarrollo de aplicaciones más productivo y que facilitase la creación de aplicaciones más complejas. Es cuando aparecen los denominados marcos de aplicaciones (application frameworks) tales como MFC (Microsoft Fundation Classes). Con ellas, surge el término de encapsulación de las funciones del API de Windows. Estas librerías permitían un rápido diseño de, por ejemplo, un cuadro de diálogo modal con estilo en tres dimensiones y dos botones, que podía ser llamado con una única sentencia.

Pero, además, se incorporaba también la potencia de la programación orientada a objetos y todo lo que ello suponía. Los proyectos podían ser divididos en miniproyectos de diseño de ciertos componentes visuales y algorítmicos que luego podían ser fácilmente enlazados en la aplicación final. Además se permitía el prototipado creando simples elementos de presentación que serían unidos al código diseñado con posterioridad. Y una importante característica de la programación orientada a objetos como es la herencia, permitía la creación de nuevos componentes dentro de la jerarquía de clases prediseñada.

Empiezan a aparecer por entonces nuevos términos como el de RAD (Rapid Application Development), pero sin duda el más conocido es el de "Programación Visual". Este fue acuñado por Microsoft Corporation(r) en 1991 con la aparición de la primera herramienta de este tipo, Visual BasicTM. Se trataba de un entorno de desarrollo visual de cuarta generación (4GL) que además integraba el editor tradicional, un compilador de Basic y un depurador con un modelo de programación dirigida por eventos, aparte de herramientas visuales que permitían un rápido y fácil desarrollo de software en entorno Windows.

La programación visual orientada a objetos es ya una realidad. Las posibilidades que se ofrecen son insospechadas. Crear un objeto o instancia de una clase (control) es tan fácil como arrastrar y soltar un control en el formulario. La particularización de un determinado objeto se consigue asignando valores a sus propiedades. El código asociado a un determinado objeto es automáticamente encapsulado a él.

Con la nueva generación de herramientas RAD, entre las que se incluye Microsoft Visual Basic 4.0, se presta principal atención al desarrollo rápido de aplicaciones en sí mismo (RAD), reutilización de componentes y la conectividad y escalabilidad de bases de datos.

Los actuales entornos RAD permiten no solo el desarrollo de aplicaciones finales sino también el diseño de prototipos eficientes, y ofrecen:

1. Entorno de desarrollo visual, que incluye, entre otros, el menú propio del entorno, la paleta de componentes, la ventana de propiedades (que refleja las propiedades asociadas al control activo) y el editor de código.
2. Controles de alto nivel, que suelen ser de dos tipos: los incorporados en la propia herramienta y aquellos externos (VBX y OCX principalmente) que pueden ser añadidos. Estos controles se disponen en forma de iconos en una paleta de componentes.
3. Acceso directo a las distintas partes del código asociado a cada uno de los objetos, y que se encuentra en la ventana del editor de código.

Las aplicaciones diseñadas se basan en entornos gráficos de usuario (GUI) fáciles de manejar y con el siguiente proceso de desarrollo:

1. Creación de un formulario vacío que contendrá los componentes del interfaz de la aplicación (controles).
2. Selección de los componentes apropiados del conjunto de los disponibles en la paleta de componentes en forma de iconos. Estos pueden luego ser redimensionados una vez colocados en el formulario.
3. Particularización de las propiedades de los controles en base a los requisitos de la aplicación.
4. Escritura de código para los distintos eventos significativos asociados al control. Este código se estructura dentro de cada procedimiento del control según el lenguaje de programación utilizado, y se pueden incluir sentencias propias del lenguaje y referencias a métodos y procedimientos del propio control o de otros controles, estén o no en el mismo formulario.
5. Ejecuciones de prueba dentro del entorno de desarrollo.
6. Modificación de la aplicación durante el proceso de desarrollo hasta la generación de un fichero ejecutable (.EXE) definitivo.
7. Utilización de herramientas de apoyo para depurar y refinar el código.

La reutilización de componentes es uno de los avances más significativos dentro de esta tecnología. Es posible realizar aplicaciones muy potentes enlazando controles de alto nivel ya diseñados y en donde el desarrollador sólo debe preocuparse de organizar el enlace entre ellos dentro de su aplicación. Estos componentes pueden generarse a partir de una DLL creada con un compilador de C/C++, como es el caso de los VBX, o pueden generarse dentro de la propia herramienta, como ocurre con los servidores OLE de Visual Basic. Estos componentes suelen diseñarse utilizando características de programación orientada a objetos, pues ofrece ventajas tales como: ciclos de desarrollo más cortos, facilidad en el mantenimiento y en el control de versiones, facilidad en la compartición de código con otros componentes, etc..

Existe la posibilidad de que varios eventos de un mismo control o de diferentes controles compartan un mismo procedimiento o función codificada en la aplicación. Además es posible utilizar funciones externas incluidas en una DLL, para ello basta con definir en el código de la aplicación el prototipo de dicha función. Estas DLL pueden haber sido diseñadas con cualquier compilador, y en ellas se incluyen las propias de Windows (GDI, KERNEL y USER) que permiten hacer uso de las funciones API del entorno Windows.

Las herramientas RAD permiten crear aplicaciones con bases de datos. La conexión con una base de datos se puede efectuar de dos formas:

1. directamente, utilizando el motor de dicha base de datos.
2. a través de enlaces remotos (vía ODBC).

En la aplicación se pueden utilizar dos alternativas:

1. controles específicos en el propio formulario.

2. objetos en el código.

Los controles de bases de datos incluidos en un formulario disponen de propiedades que permiten especificar la tabla o la base de datos con la que se enlazará. Los objetos incluidos en el código disponen de métodos que enlazan de igual forma con bases de datos. En ambos casos es posible indicar si dicho enlace es directo (a través del motor de la base de datos) o es remoto. Ambas alternativas también contemplan el acceso a los datos a través de los métodos que implementan o utilizando búsquedas con sentencias SQL.

Además de los controles propios de bases de datos, adicionalmente se utilizan otros como cuadros de texto, etiquetas, listas desplegables, controles de tipo tabla (grids), etc. para la visualización de los resultados de lectura o para permitir la introducción de los datos a escribir en la base de datos.

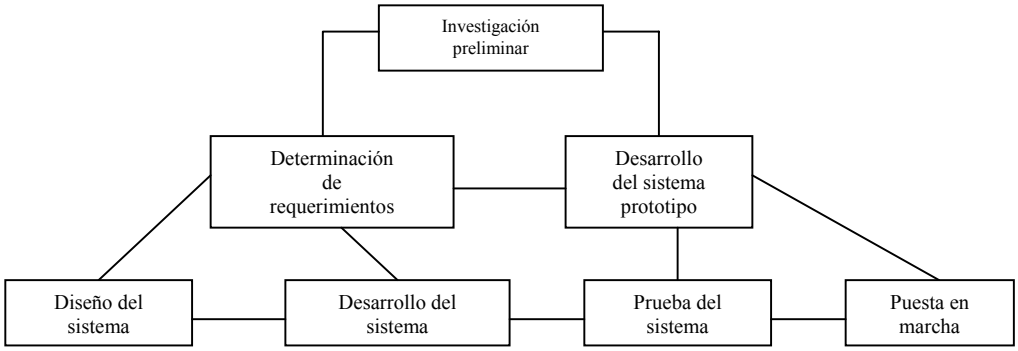
C. SISTEMAS E INDUSTRIA DEL SOFTWARE

I. Ciclo de vida del desarrollo de sistemas

1. Fases del desarrollo de software

El desarrollo de sistemas es un proceso que consiste en dos etapas principales de análisis y diseño de sistemas; comienza cuando la gerencia, o en algunas ocasiones el personal de desarrollo de sistemas, se da cuenta de que cierto sistema del negocio necesita mejorarse. El ciclo de vida del desarrollo de sistemas es el conjunto de actividades de los analistas, diseñadores y usuarios, que necesitan llevarse a cabo para desarrollar y poner en marcha un sistema de información. Se debe tener presente que en la mayoría de las situaciones del negocio, las actividades están íntimamente relacionadas y son inseparables. El ciclo de vida del desarrollo de sistemas consiste en las siguientes actividades:

1. Investigación preliminar
2. Determinación de requerimientos
3. Desarrollo de sistema prototipo
4. Diseño de sistema
5. Desarrollo de software
6. Prueba de los sistemas
7. Puesta en marcha



- ***Análisis preliminar (enunciado general del problema, propósito del sistema, requerimientos, principales entradas y salidas, diagrama de bloques del sistema, estimación de costos y beneficios y plan preliminar de desarrollo)***

Investigaciones preliminares

¿Cuantas veces se está en situaciones en donde se pregunta si no existe una mejor manera de hacer algo? Por ejemplo, abrir una tienda departamental adicional que creará una necesidad para nuevos procedimientos de facturación, cuando un alto porcentaje de clientes utiliza la cuenta de crédito de esta compañía y compra en todas las tiendas. Se puede iniciar una petición por muchas razones, pero la clave es que alguien, ya sea gerente, un empleado o un especialista de sistemas, inicie un requerimiento para recibir ayuda de un sistema de información. Cuando ese requerimiento se realiza, la primera actividad de sistemas, es decir, la investigación preliminar, se inicia. Esta actividad tiene tres partes: clarificación de requerimiento, estudio de la factibilidad y aprobación del

requerimiento. El resultado será aprobar el requerimiento para la atención posterior o rechazarlo como no factible para un desarrollo futuro.

Estudio de Factibilidad

Un resultado importante de la investigación preliminar es la determinación de que el sistema requerido es factible. Existen tres aspectos en el estudio de factibilidad de la investigación preliminar:

1. Factibilidad técnica. ¿Puede realizarse el trabajo para el proyecto con el equipo actual, tecnología de software y el personal disponible? Si se requiere nueva tecnología, ¿qué probabilidades hay de que pueda desarrollarse?
2. factibilidad económica. ¿Existen suficientes beneficios en la creación del sistema para hacer que los costos sean aceptables? O, en forma inversa, ¿son tan altos los costos como para que el proyecto no deba llevarse a cabo?
3. Factibilidad operativa. ¿Se utilizará el sistema si se desarrolla y pone en marcha? Habrá resistencia de los usuarios, que los posibles beneficios reducirán del sistema.

El estudio de factibilidad se lleva a cabo con un pequeño grupo de gente, familiarizada con las técnicas de los sistemas de información, que entienden la parte de la empresa que será afectada por el proyecto y tienen los conocimientos suficientes del proceso de análisis y diseño de sistemas.

- ***Análisis detallado (técnicas de recolección de información, determinación de los requerimientos detallados, formalización de los requerimientos, especificación de alcances y limitaciones del sistema)***

Clarificación del requerimiento

En las empresas muchos requerimientos de los empleados y usuarios no están establecidos claramente; por lo tanto, antes de que pueda considerarse la investigación del sistema, el proyecto requerido debe examinarse para determinar precisamente lo que desea la empresa. Una simple llamada telefónica puede ser suficiente si la persona que requiere el servicio tiene una idea clara, pero no sabe cómo establecerla. Por otro lado, la persona que hace el requerimiento puede estar simplemente pidiendo ayuda sin saber qué es lo que está mal o por qué existe un problema. La clarificación del problema es este caso, antes de poder llagar a otro paso, el requerimiento de proyecto debe estar claramente establecido.

Aprobación del requerimiento

No todos los proyectos requeridos son deseables o factibles. Sin embargo, aquellos que son tanto factibles como deseables deben anotarse para tomarlos en cuenta. En algunos casos, el desarrollo puede comenzar inmediatamente, pero en la mayor parte, los miembros del departamento de sistemas están ocupados en otros proyectos que se encuentran en marcha. Cuando esto sucede, la gerencia decide que proyectos son más importantes y entonces los programa. Después de que se aprueba la requisición de un proyecto, se estima su costo, la prioridad, el tiempo de terminación y los requerimientos del personal que se utilizan. Posteriormente, cuando se terminan algunos proyectos anteriores, puede iniciarse el desarrollo de la aplicación propuesta. En este momento, comienza la recabación de datos y la determinación de los requerimientos.

Determinación de requerimientos

El punto clave de análisis de sistemas se consigue al adquirir un conocimiento detallado de todas las facetas importantes dentro del área de negocios que se investiga. (Por esta razón, a menudo esta actividad se conoce como *investigación detallada*.) Los analistas, al trabajar con los empleados y gerentes, deben estudiar el proceso que actualmente se efectúa para contestar estas preguntas clave:

1. ¿Qué se está haciendo?
2. ¿Cómo se está haciendo?
3. ¿Qué tan frecuentemente ocurre?
4. ¿Qué tan grande es la cantidad de transacciones o decisiones?
5. ¿Qué tan bien se lleva acabo la tarea?
6. ¿Existe algún problema?
7. ¿Si el problema existe, qué tan serio es?
8. ¿Si el problema existe, cuál es la causa principal?

Para contestar estas preguntas, los analistas de sistemas hablarán con diferentes personas para recabar los detalles en relación con el proceso, así como sus opiniones sobre las causas por las cuales suceden las cosas de esa manera y algunas ideas en relación a modificarlas. Se utilizan cuestionarios para recopilar esta información, aplicándolos a grandes que no pueden entrevistarse en forma individual. Las investigaciones detalladas también requieren el estudio de manuales y reportes, la observación real de las actividades de trabajo y algunas veces la recabación de formas y documentos para entender completamente el proceso. Conforme se recopilan los elementos, los analistas estudian

los requerimientos de datos para identificar las características que tendrá el nuevo sistema, incluyendo la información que el sistema debe producir y las características operativas, como son controles de procesamiento, tiempos de respuesta y métodos de entrada y salida.

- ***Diseño preliminar (definición de restricciones, diseño de la arquitectura del sistema, definición de interfaces entre módulos)***

Desarrollo del sistema prototipo

La **preparación de prototipos** es el proceso de crear, desarrollar y refinar un modelo funcional del sistema final. Se puede crear un **modelo prototipo preliminar** durante la etapa de definición del problema. Un miembro del equipo de reconocimiento -suponga que se trata de un especialista en el procesamiento de datos- puede construir un modelo de este tipo que muestre la composición de las pantallas y los formatos de los informes. Durante una sesión de requerimientos, otros miembros del equipo y usuarios del futuro sistema examinan esta muestra en la forma con el constructor del modelo entiende en principio el problema y los resultados que debe producir el sistema. En este momento puede iniciarse un proceso de refinación si los usuarios señalan omisiones y equivocas. Durante este proceso de refinación, cuyo objetivo es definir la necesidad que existe, uno o más miembros del equipo pueden utilizar una computadora personal y un **paquete de programas de prototipos** a fin de crear una serie de pantallas en la computadora personal. Estas pantallas no son las salidas que producen los programas ya terminados, pero pueden parecerse mucho a esos resultados. Es posible exhibir en el monitor de la computadora, como una secuencia de diapositivas, menús de captura de datos, la interfaz con el usuario debe servir para buscar, consultar y manipular datos y el formato de los informes de salida. Por ejemplo, se pueden simular los resultados de una serie de selecciones hechas en menús para que los usuarios tengan una idea más clara de la forma como el constructor o los constructores del sistema están interpretando el problema. Si los usuarios no están convencidos de lo que se exhibe define con precisión sus necesidades, pueden modificar fácilmente las plantillas prototipo hasta que estén satisfechos. La creación de un modelo preliminar de prototipo en este punto produce varios beneficios: los usuarios pueden ver que se está avanzado, se les motiva para que participen activamente en la definición del problema, se mejora la comunicación entre todas las partes interesadas y se aclaran los equívocos en una etapa temprana del estudio de sistemas, antes de que se conviertan en costosos errores. Como se acaba de ver, puede ser necesario un proceso repetitivo (o *interactivo*) para terminar el paso de definición del problema. No existe un procedimiento definido que se deba seguir antes de que se pueda iniciar el análisis detallado del sistema. Un alto ejecutivo puede creer que existen diferencias de información. Puede preparar una declaración general de los objetivos y nombrar a un gerente para que realice un reconocimiento. Pueden realizarse varias sesiones de requerimientos para traducir los deseos generales a objetivos más específicos. Asimismo, pueden crearse y refinarse modelos preliminares de prototipo; se puede ampliar o reducir el alcance del estudio y es posible también que cambien los objetivos conforme se reúnan los datos. Una vez que parezca haberse logrado la aprobación en cuanto a la definición del problema, el equipo de reconocimiento deberá poner la definición detallada *por escrito* y enviarla a todas las personas interesadas, las cuáles deberán aprobarla también por escrito. Si persisten diferencias, deberán resolverse en sesiones adicionales de requerimientos. Hay quienes se impacientan con los “retrasos” en el desarrollo del sistema causados por estas sesiones adicionales. Sin embargo, las personas más prudentes saben que los retrasos verdaderamente largos y costosos se presentan cuando los usuarios descubren, ya muy avanzados el proceso del desarrollo, que el sistema diseñado no es satisfactorio por haberse pasado por alto algunos requerimientos.

Diseño del sistema

El diseño de un sistema de información produce los elementos que establecen cómo el sistema cumplirá los requerimientos indicados durante el análisis de sistemas. A menudo los especialistas de sistemas se refieren a esta etapa como en *diseño lógico*, encontraste con desarrollo del software de programas, que se conoce como *diseño físico*. Los analistas de sistemas comienzan por identificar los informes y otras salidas que el sistema producirá. A continuación los datos específicos con éstos se señalan, incluyendo su localización exacta sobre el papel, la pantalla de despliegue u otro medio. Usualmente, los diseñadores dibujan la forma o la visualización como la esperan cuando el sistema esta terminado. El diseño del sistema también describe los datos calculados o almacenados que se introducirán. Los grupos de datos individuales y los procedimientos de calculo se describen con detalle. Los diseñadores seleccionan las estructuras de los archivos y los dispositivos de almacenamiento, como son discos magnéticos, cintas magnéticas o incluso archivos en papel. Los procedimientos que ellos escriben muestran cómo se van a procesar los datos y a producir la salida. Los documentos que contienen las especificaciones de diseño utilizan muchas formas para representar los diseños, diagramas, tablas y símbolos especiales, algunos de los cuales el lector puede haber utilizado ya y otros que pudieran ser totalmente nuevos. La información del diseño detallado se pasa al grupo de programación para que pueda comenzar el desarrollo del software. Los diseñadores son responsables de proporcionar a los programadores las especificaciones completas y escritas con claridad, que establezcan lo que

debe hacer el software. Conforme comienza la programación, los diseñadores están pendientes para contestar preguntas, esclarecer ideas confusas y manejar los problemas que confronten los programadores cuando utilicen las especificaciones de diseño.

- ***Construcción (programación estructurada u orientada a objetos, pruebas de componentes)***

Desarrollo del Software

Los desarrollares del software pueden instalar o modificar; por ejemplo, software comercial que se haya comprado, o pueden escribir programas nuevos diseñados a la medida. La decisión de qué se va a hacer depende del costo de cada una de las opciones, el tiempo disponible para describir el software y la disponibilidad de programadores. En forma usual, en las grandes empresas los programadores de computadoras (o la combinación de analistas-programadores) son parte del grupo profesional permanente. Las compañías más pequeñas en donde los programadores permanentes no se han contratado, pueden obtener servicios externos de programación con base en un contrato. Los programadores también son responsables de documentar el programa e incluir los comentarios que expliquen tanto cómo y por qué se utilizo cierto procedimiento conforme se codifico de cierta forma. La documentación es esencial para probar el programa y darle mantenimiento una vez que la aplicación se ha puesto en marcha.

- ***Pruebas (elaboración del plan de pruebas, preparación de casos, establecimiento de criterios de revisión, elaboración de las pruebas, evaluación de resultados)***

Prueba de los sistemas

Durante la prueba, el sistema se utiliza en forma experimental para asegurar que el software no falle; es decir, Que corra de acuerdo a sus especificaciones y a la manera que los usuarios esperan que lo haga. Se examinan datos especiales de prueba en la entrada del procesamiento y los resultados para localizar algunos problemas inesperados. Puede permitirse también a un grupo limitado de usuarios que utilice el sistema, de manera que los analistas puedan captar si tratan de utilizarlo en forma no planeadas. Es preferible detectar cualquier anomalía antes de que la empresa ponga en marcha el sistema y dependa de él. En muchas compañías la prueba se lleva a cabo por personas diferentes a aquellos que los escriben en forma original; es decir si se utilizan personas que no conocen como se diseñaron ciertas partes de los programas, se asegura una mayor y más completa prueba, además de ser imparcial, lo que da a un software más confiable.

- ***Instalación (manual de usuario, ayudas interactivas, capacitación, corridas de prueba)***

Puesta en marcha

Cuando el personal de sistemas verifica y pone en uso el nuevo equipo, entrena al personal (usuario); instala la nueva aplicación y constituye los archivos de datos que se necesiten, entonces el sistema está puesto en marcha. De acuerdo con el tamaño de la empresa que empleará la aplicación y el riesgo asociado con su uso, los desarrolladores del sistema pueden escoger una prueba piloto para la operación del sistema solamente en un área de la compañía; por ejemplo, en un departamento o sólo con una o dos personas. A veces correrán en forma paralela tanto el sistema anterior como el nuevo para comparar los resultados de ambos; en otras situaciones, los desarrolladores pararán por completo el sistema anterior un día y al siguiente empezarán a utilizar el nuevo. Como se puede apreciar, cada estrategia para la puesta en marcha tiene sus méritos, que dependen de la situación del negocio considerado. Sin importar la estrategia para la puesta en marcha que se haya utilizado, los desarrolladores tendrán que asegurarse que el uso inicial del sistema esté libre de problemas. Una vez instalada, con frecuencia la aplicación se utiliza por muchos años; sin embargo, tanto la empresa como los usuarios cambiarán, y el medio ambienta será diferente también a través del tiempo. Por lo tanto, la aplicación indudablemente necesitará mantenimiento; es decir, se harán cambios y modificaciones al software, y a los archivos o procedimientos para cubrir los requerimientos nuevos de los usuarios. Los sistemas de la empresa y el medio ambiente de los negocios están en continuo cambio. Los sistemas de información deben mantenerse de la misma forma; es este sentido, la propuesta en marcha es un proceso continuo.

- ***Mantenimiento (correctivo, adaptativo, perfectivo, pruebas de regresión, técnicas de mantenimiento)***

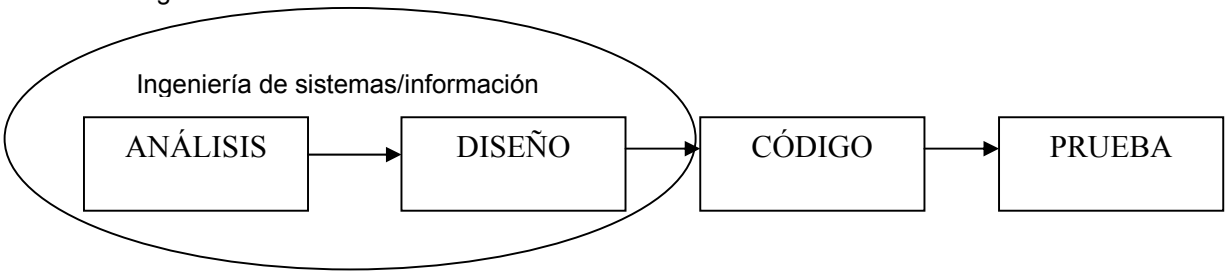
La fase de mantenimiento se centra en los cambios que va a sufrir el software a lo largo de su vida útil. Como ya hemos dicho, estos cambio pueden deberse a la corrección de errores, a cambios en el entorno inmediato del software o a cambios en los requisitos del cliente, dirigidos normalmente a ampliar el sistema.

La fase de mantenimiento vuelve a aplicar los pasos de las fases de definición y de desarrollo, pero en el contexto de un software ya existente y en funcionamiento.

2. Ciclos de vida de desarrollo de sistemas

- ***En cascada***

El modelo lineal secuencial para la ingeniería del software, llamado algunas veces ciclo de vida básico o modelo en cascada, sugiere un enfoque sistemático secuencial del desarrollo del software que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento. El modelo lineal secuencial acompaña a las actividades siguientes:



Ingeniería y modelado de sistemas/información
El trabajo comienza estableciendo requisitos de todos los elementos del sistema y asignando al software algún subgrupo de estos requisitos. Esta visión del sistema es esencial cuando el software se debe interconectar con otros elementos como hardware, personas y bases de datos. La ingeniería y el análisis de sistemas acompaña a los requisitos que se recogen en el nivel del sistema con una pequeña parte de análisis y de diseño.
Análisis de los requisitos del software
El proceso de reunión de requisitos se intensifica y se centra en el software. El ingeniero (analista) del software debe comprender el dominio de información del software, así como la función requerida, comportamiento, rendimiento e interconexión. El cliente documenta y repasa los requisitos del sistema.
Diseño
El diseño del software es un proceso de mucho pasos que se centra en cuatro atributos distintos de un programa: estructura de datos, arquitectura del software, representaciones de interfaz y detalle procedimental (algoritmo). El proceso de diseño traduce requisitos en una representación del software que se pueda evaluar por calidad antes de que comience la generación del código. El diseño se documenta y se hace parte de la configuración del software.
Generación de código
El diseño se debe traducir en una forma legible por la maquina. El paso de generación de código lleva a cabo esta tarea. Si se lleva a cabo un diseño detallado, la generación de código se realiza mecánicamente.
Pruebas
Una vez que se ha generado un código, comienzan las pruebas del programa. El proceso de pruebas se centra en los procesos lógicos internos del software, asegurando que todas las sentencias se han comprobado y el sentirse seguro de que la entrada definida produzca resultados reales de acuerdo con los resultados requeridos.
Mantenimiento
El software indudablemente sufrirá cambios después de ser entregado al cliente (una excepción posible es el software empotrado). Se producirán cambios porque se han encontrado errores o porque el software debe adaptarse para acoplarse a los cambios de su entorno externo o porque el cliente requiere mejoras funcionales o de rendimiento. El mantenimiento vuelve a aplicar cada una de las fases precedentes a un programa ya existente y no a uno nuevo. El modelo lineal secuencial es el paradigma mas antiguo y mas extensamente utilizado en la ingeniería del software. Sin embargo, la critica ha puesto en duda su eficacia. Entre los problemas que se encuentran algunas veces en el modelo lineal secuencial estn:

- Los proyectos reales raras veces siguen el modelo secuencial que propone el modelo.

- A menudo es difícil que el cliente exponga explícitamente todos lo requisitos.
- El cliente debe tener paciencia. Una versión del programa no estarán disponibles hasta que el proyecto este my avanzado.
- Los responsables del desarrollo del software siempre se retrasan innecesariamente.

- ***Espiral***

El modelo en espiral es un modelo de proceso de software evolutivo que acompaña la naturaleza interactiva de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial. Proporciona el potencial para el desarrollo rápido de versiones incrementales del software. Durante las primeras iteraciones, la versión incremental podría ser un modelo en papel o un prototipo. Durante las ultimas iteraciones se producen versiones cada vez más completas de ingeniería del sistema. El modelo en espiral se divide en un numero de actividades estructurales, también llamadas regiones de tareas. Generalmente, existen entre tres y seis regiones de tareas:

Comunicación con el cliente
Las tareas requeridas para establecer comunicación entre el desarrollador y el cliente.
Planificación
Las tareas requeridas para definir recursos, el tiempo y otras informaciones relacionadas con el proyecto.
Análisis de riesgos
Las tareas requeridas para evaluar riesgos técnicos y de gestión.
Ingeniería
Las tareas requeridas para construir una o más representaciones de la aplicación.
Construcción y adaptación
Las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario.
Evaluación del cliente
Las tareas requeridas para obtener la reacción del cliente según la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implementada durante la etapa de instalación. A diferencia del modelo de proceso clásico que termina cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora. Un proyecto de desarrollo de conceptos comienza en el centro de la espiral y continuará hasta que se completa el desarrollo del concepto. el modelo en espiral utiliza la construcción de prototipos como mecanismos de reducción de riesgos, pero lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto.

II. Metodologías para el desarrollo de sistemas

1. Enfoque funcional (Yourdon)

- ***Análisis estructurado (diagrama de flujo de datos, diccionario de datos, miniespecificaciones, diagramas de transición de estado)***
- ***Diseño estructurado (diagramas de estructura, acoplamiento, cohesión, análisis de transformaciones, análisis de transacciones)***

Análisis.
El análisis estructurado, como todos los demás métodos de análisis de requisitos, es una actividad de construcción de modelos. Mediante una notación que es única de este método, se crean modelos que reflejan el flujo y el contenido de la información (datos y control); se parte el sistema funcionalmente y, según los distintos comportamientos, se establece la esencia de lo que se debe construir.

La tarea del análisis de sistemas, conlleva más que sólo realizar análisis de requisitos, pero es en eso donde se focalizará la discusión.

Una de las principales labores del analista es descubrir detalles y documentar la política de un negocio que pudieran existir sólo en forma implícita, "transmitidas de generación en generación" por los usuarios, nunca documentadas formalmente. El analista debe distinguir entre síntomas, problemas del usuario y causas. Con sus conocimientos de la tecnología de los computadores, el analista debe ayudar al usuario a explorar aplicaciones novedosas y más útiles de éstos así como nuevas formas de hacer negocios. Aunque muchos de los sistemas antiguos sólo se limitaban a perpetuar el negocio original del usuario, pero a velocidades electrónicas, hoy en día los analistas se enfrentan al

desafío de ayudar al usuario a encontrar productos y mercados radicalmente innovadores, con la ayuda del computador.

Diseño.

El diseño de software es un proceso mediante el que se traducen los requisitos en una representación del software. Inicialmente, la representación describe una visión holística del software. Posteriores refinamientos conducen a una representación de diseño que se acerca mucho al código fuente.

En el diseño se realizan dos pasos. El diseño preliminar se centra en la transformación de los requisitos en los datos y arquitectura del software. El diseño detallado se ocupa del refinamiento de la representación arquitectónica que lleva a una estructura de datos detallada y a las representaciones algorítmicas del software.

Dentro del contexto de los diseños preliminar y detallado, se llevan a cabo varias actividades de diseño diferentes. Además del diseño de datos, del diseño arquitectónico y del diseño procedimental, muchas aplicaciones requieren de un diseño de la interfaz. El diseño de la interfaz establece la disposición y los mecanismos para la interacción hombre máquina (no cubierto por las herramientas del diseño estructurado).

Fundamentos del Análisis y Diseño.

Abstracción.

Cuando se considera una solución modular para cualquier problema, pueden formularse muchos niveles de abstracción. En el nivel superior de abstracción, se establece una solución en términos amplios, usando el lenguaje del entorno del problema. En los niveles inferiores de abstracción se toma una orientación más procedimental. La terminología orientada al problema se acompaña con una terminología orientada a la implantación, en un esfuerzo para establecer una solución. Por último, en el nivel más bajo de abstracción, se establece la solución de forma que pueda implementarse directamente.

Refinamiento

El refinamiento sucesivo es una primera estrategia de diseño descendente (propuesta por Niklaus Wirth). Un programa se desarrolla en niveles sucesivos de refinamiento de los detalles procedimentales. Se desarrolla una jerarquía descomponiendo una declaración macroscópica de una función en forma sucesiva hasta que se llega a las sentencias del lenguaje de programación. Cada paso de refinamiento implica algunas decisiones de diseño. Es importante que el programador sea consciente de sus decisiones y de la existencia de soluciones alternativas.

Modularidad

Se ha dicho que modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable. El software monolítico (compuesto por sólo un módulo) no puede ser fácilmente abarcado por un lector. El número de caminos de control, la expansión de referencias, el número de variables y la complejidad global podrían hacer imposible su correcta comprensión.

La modularidad se deriva naturalmente de un principio elemental para manejar la complejidad: divide y vencerás.

Diseño Modular Efectivo.
La calidad del diseño debe ser una meta para el diseñador. El diseño estructurado ofrece guías para apoyar al diseñador a determinar módulos, y sus interconexiones, que mejor realizarán los requerimientos especificados por el analista. Las dos reglas más importantes son las referentes al acoplamiento y la cohesión.

Cohesión.
Grado en el cuál los componentes de un módulo (típicamente las instrucciones individuales que lo conforman) son necesarios y suficientes para llevar a cabo una sola función bien definida. En la práctica, esto significa que el diseñador debe asegurarse de no fragmentar los procesos esenciales en módulos, y también debe asegurarse de no juntar procesos no relacionados en módulos sin sentido. Los mejores módulos son aquellos que son funcionalmente cohesivos (es decir, módulos en los cuales cada instrucción es necesaria para poder llevar a cabo una tarea bien definida). Los peores módulos son los que son coincidentalmente cohesivos (es decir, donde sus instrucciones no tienen una relación significativa entre uno y otro).

Los grados de cohesión, de menor a mayor son:

a. Cohesión Coincidental. No existe una relación significativa entre los elementos del módulo.

b. Cohesión Lógica. La relación entre los elementos del módulo está basada en obtener ventajas en el procesamiento, por ejemplo, todos manipulan el mismo dato. Normalmente esto implica tener un código truculento o compartido, que degrada los propósitos de un buen diseño.

c. Cohesión Temporal. Los elementos del módulo constituyen un conjunto que se ejecuta secuencialmente en un punto fijo en el tiempo. Aunque tiende, a veces, a confundirse con la cohesión lógica, la diferencia está en que este tipo de módulo s más simple y se ejecuta sin la intervención de otras aplicaciones.

d. Cohesión Comunicacional. Los elementos del módulo hacen referencia al mismo conjunto de datos. Aquí se presenta un grado "aceptable" de cohesión.

e. Cohesión Secuencial. Implica que la salida de un elemento es la entrada para el próximo.

f. Cohesión Funcional. Aquí, todos los elementos del módulo están orientados a la realización de una función única.

Acoplamiento.
Grado en el cuál los módulos se interconectan o se relacionan entre ellos. Entre más fuerte sea el acoplamiento entre módulos en un sistema, más difícil es implantarlo y mantenerlo, pues entonces se necesitará un estudio cuidadoso para la modificación de algún módulo o módulos. En la práctica, esto significa que cada módulo debe tener interfaces sencillas y limpias con otros, y que se debe compartir un número mínimo de datos entre módulos. También significa que un módulo dado no debe modificar la lógica interna o los datos de algún otro módulo; lo que se conoce como una conexión patológica.

Tamaño del Módulo.
De ser posible, cada módulo debe ser lo suficientemente pequeño como para caber en una sola página (o para que se pueda desplegar en una sola pantalla). Desde luego, a veces no es posible determinar qué tan grande va a ser un módulo hasta haberlo escrito, pero las actividades iniciales de diseño a menudo darán al diseñador una buena pista de que el módulo será grande o complejo. Si es así, debe subdividirse en uno o más niveles de submódulos.

Alcance del control.
El número de subordinados inmediatos que un módulo administrador puede llamar se conoce como el alcance del control. Un módulo no debe poder llamar a más de una media docena de módulos de nivel inferior. La razón es evitar la complejidad: si el módulo tuviera, por ejemplo, que llamar a 25 módulos de nivel inferior, entonces seguramente contendrá tanta lógica compleja que nadie lo entenderá (un sin fin de if-then anidados). La solución es introducir un nivel intermedio de módulos administradores, como haría un administrador de una organización humana.

Alcance del efecto/alcance del control.
Esta regla sugiere que cualquier módulo afectado por el resultado de alguna decisión debe ser subordinado (aunque no necesariamente un subordinado inmediato) del módulo que toma la decisión. Es un tanto análogo a la regla de administración que dice que cualquier empleado afectado por los resultados de la decisión de algún administrador (es decir, dentro del alcance de efecto de la decisión), debe estar dentro del alcance de control del administrador (es decir trabajando entre la jerarquía de personas que se reportan con el administrador). Violar esta regla en un ambiente de diseño estructurado usualmente lleva a un paso innecesario de banderas y condiciones (lo cual incrementa el acoplamiento entre módulos), la toma redundante de decisiones o (en el peor de los casos) conexiones patológicas entre módulos.

Parsimonia.
Se refiere a la economía de recursos que se emplean para la obtención de un resultado. Esto es, sólo se debe realizar lo que se pide. Mientras mayor la parsimonia, mejor el diseño.

Manejo Autónomo de Errores.
Los módulos deben tener la capacidad de manejar sus propias condiciones de error, tanto en la detección como en la corrección de los mismos. De no ser así, el manejo de banderas (flags) de control y la transmisión de datos erróneos a otros módulos aumentará considerablemente el acoplamiento.

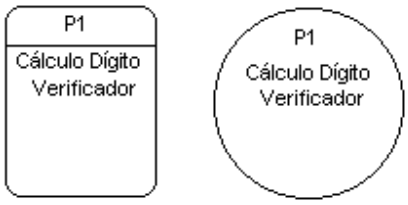
Diagramas de Flujo de Datos.
Los diagramas de flujos de datos también son llamados Carta de Burbujas, DFD, Diagramas de burbujas, modelo de proceso, diagrama de flujo de trabajo o modelo de función en la literatura computacional.

A medida que la información se mueve a través del software, es modificada por una serie de transformaciones. El DFD es una técnica gráfica que representa el flujo de la información y las transformaciones que se aplican a los datos al moverse desde la entrada hasta la salida.

Componentes de un DFD.

El proceso.
Sinónimos comunes son burbuja, función o transformación.

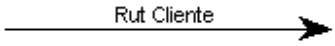
El proceso muestra una parte del sistema que transforma entradas en salidas; es decir, muestra cómo es que una o más entradas se transforman en salidas. El proceso se representa gráficamente como un óvalo o un rectángulo con esquinas redondeadas. Estas diferencias son sólo de forma, y se debe optar por alguna de ellas y utilizarla en forma consistente.



Representaciones utilizadas para procesos, la de la izquierda corresponde a la utilizada por Gane y Sarson, y la de la derecha es utilizada por Ward y Mellor, así como por Yourdon y De Marco.

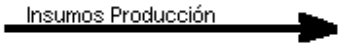
Nótese que el proceso se nombra con una palabra o frase, que intentan dar una primera aproximación de lo que hacen, por ejemplo VALIDAR ENTRADA, CONTROL TEMPERATURA, etc.

El flujo.
Un flujo se representa gráficamente por medio de una flecha que entra o sale de un proceso. El flujo se usa para describir el movimiento de bloques o paquetes de información de una parte del sistema a otra. Por ello, los flujos representan datos en movimiento, mientras que los almacenes representan datos en reposo.



Flujo de Datos, que lleva el Rut de un cliente. Se utiliza esta presentación en casi todos los formalismos propuestos.

En la mayoría de los sistemas que se modelan, los flujos realmente representarán datos, es decir, bits, caracteres, mensajes, números de punto flotante y los diversos otros tipos de información con los que se suele tratar en sistemas computarizados. Esto no significa que los DFD no sean una herramienta útil en el modelado de procesos no automatizados computacionalmente, como por ejemplo una linea de ensamblado.



Este es la representación dada por Gane y Sarson a un flujo de materiales. Con esto, se representa que se ingresan datos o materiales de tipo no computacional. Es útil en el modelamiento de procesos productivos.

Los flujos de datos tienen un nombre el que representa el significado del paquete de información que se mueve a lo largo del flujo.

Los flujos de datos pueden converger o divergir en un DFD.

El almacén.
El almacén se utiliza para modelar un conjunto de paquetes de datos en reposo. Se denota por dos líneas paralelas u otras alternativas gráficas. De modo característico, el nombre que se usa para un almacén es el plural del que se usa para los paquetes que entran y salen del almacén por medio de flujos.

Representaciones utilizadas para almacenes de datos, la de la izquierda corresponde a la utilizada por Gane y Sarson, y la de la derecha es utilizada por Ward y Mellor, así como por Yourdon y De Marco.

A menudo, los almacenes de datos se implementan como archivos o bases de datos. También pueden ser implementados en sistemas manuales como archivadores, carpetas, etc.

El Terminador.
Un terminador gráficamente se representa como un rectángulo. Los terminadores representan entidades externas con las cuales el sistema se comunica. Comúnmente un terminador es una persona o un grupo, por ejemplo una organización externa o una agencia gubernamental, o un grupo o departamento que esté dentro de la misma compañía u organización, pero fuera del control del sistema que se está modelando. En algunos casos, el terminador puede ser otro sistema.

Terminador o "External", que en este caso representa al usuario del sistema. Se utiliza esta presentación en casi todos los formalismos propuestos.

Suele ser muy fácil identificar los terminadores en el sistema que se está modelando. A veces el terminador es el usuario, que nos dice "pienso entregar los datos A, B y C al sistema y espero que éste me entregue los datos X, Y y Z". En otros casos, el usuario se considera parte del sistema y ayudará a identificar los terminadores relevantes.

Guía para la construcción de un DFD.
a. Escoger nombres con significado para los procesos, flujos, almacenes y terminadores.

b. Numerar los procesos.

c. Redibujar el DFD tantas veces como sea necesario estéticamente.

d. Evitar los DFD excesivamente complejos.

e. Asegurarse de que el DFD sea internamente consistente y que también lo sea con cualesquiera DFD relacionado con él. (evitar procesos con sólo entradas o salidas, así como flujos y procesos no etiquetados).

DFD por niveles.
Se organiza el DFD global en una serie de niveles de modo que cada uno proporcione sucesivamente más detalles sobre una porción del nivel anterior. Esto es análogo a la organización de mapas en un atlas.

El DFD de primer nivel consta sólo de una burbuja, que representa el sistema completo; los flujos de datos muestran las interfaces entre el sistema y los terminadores externos (junto con los almacenes externos que pudiera haber). Este DFD especial se conoce como Diagrama de Contexto.

El DFD que sigue del diagrama de Contexto se conoce como la figura 0. Representa la vista de más alto nivel de las principales funciones del sistema, al igual que sus principales interfaces.

Ejemplo de un diagrama de contexto.

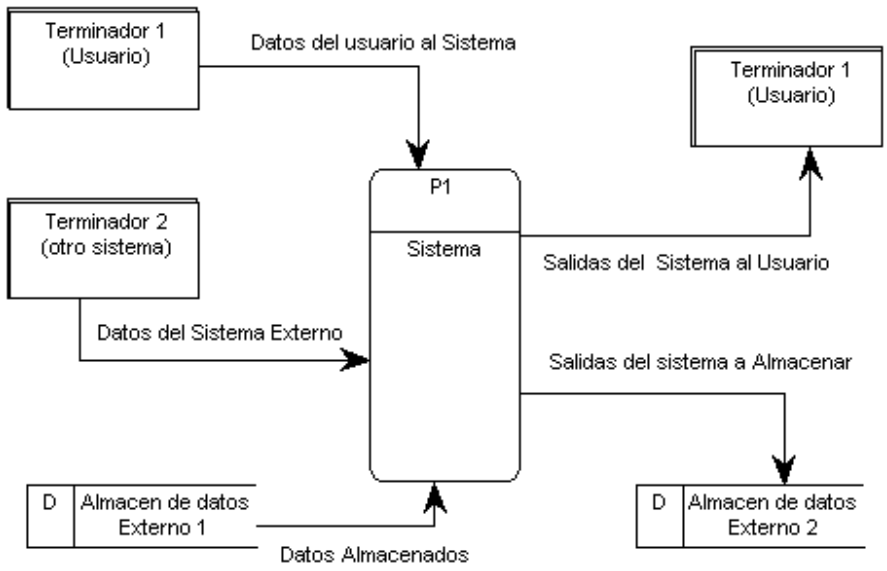


Diagrama nivel 0. Aquí se presenta la primera descomposición funcional del sistema.

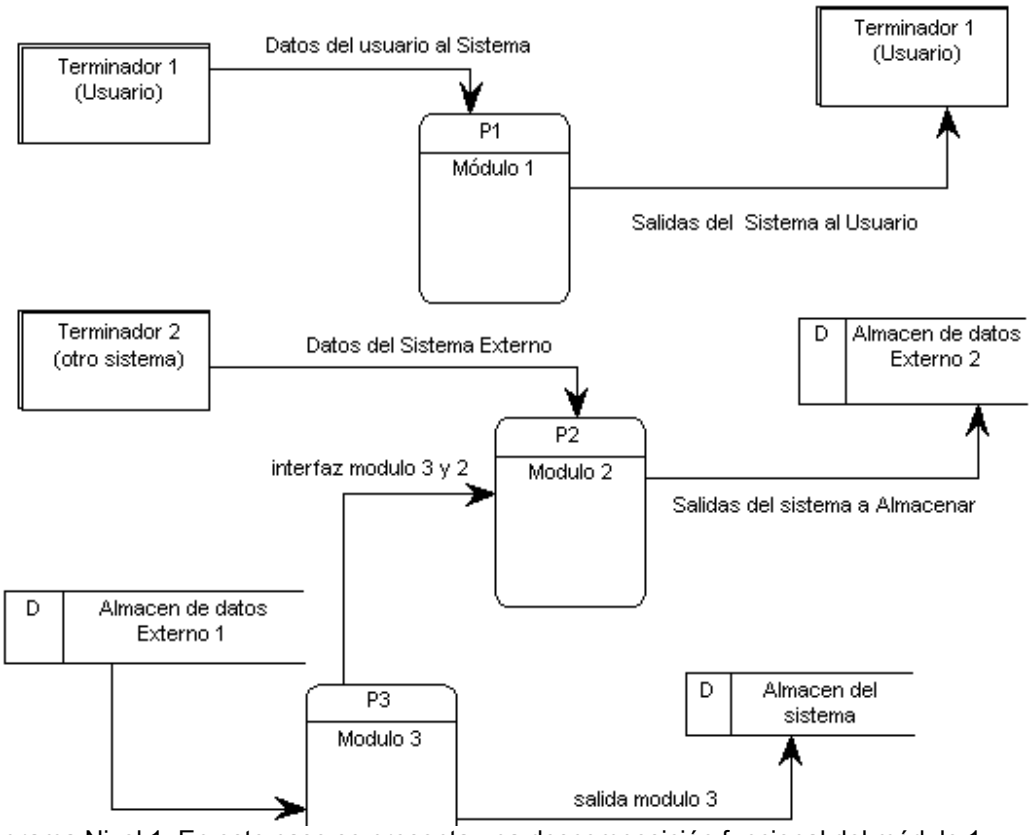


Diagrama Nivel 1. En este caso se presenta una descomposición funcional del módulo 1.

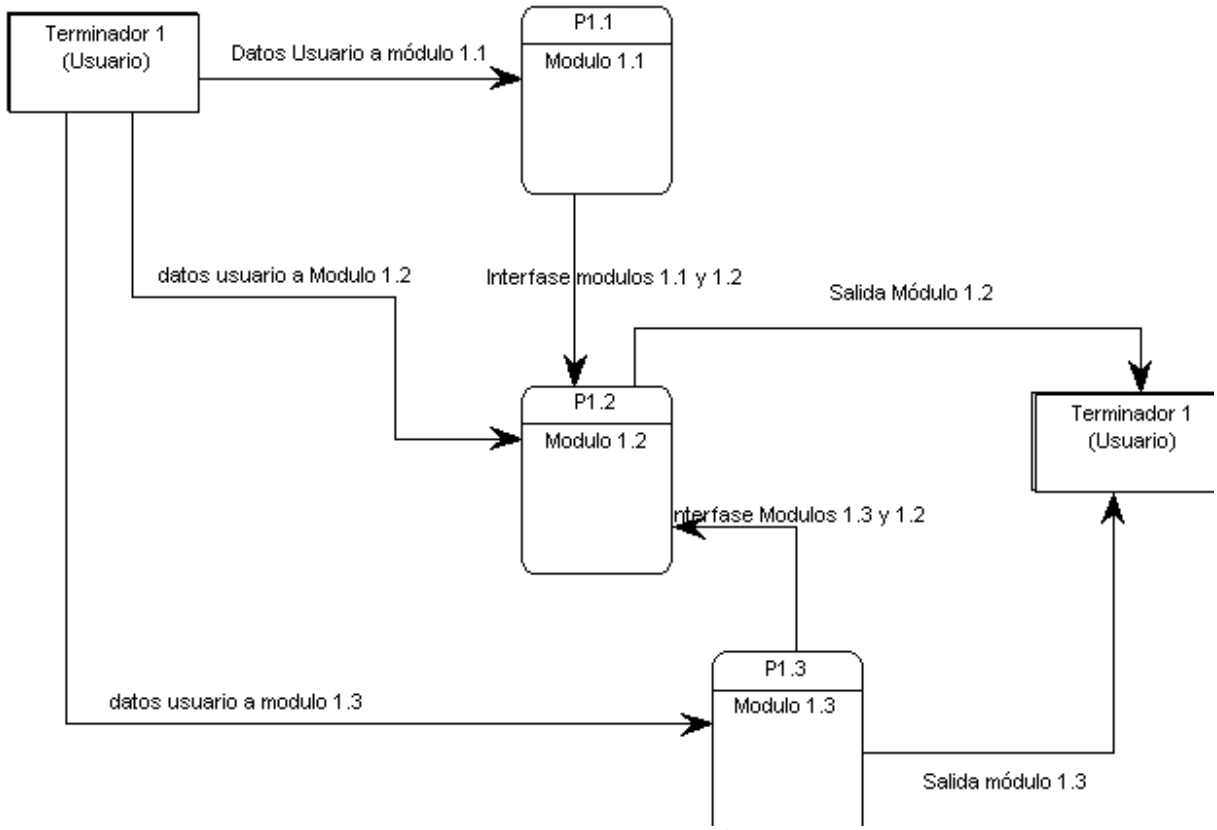
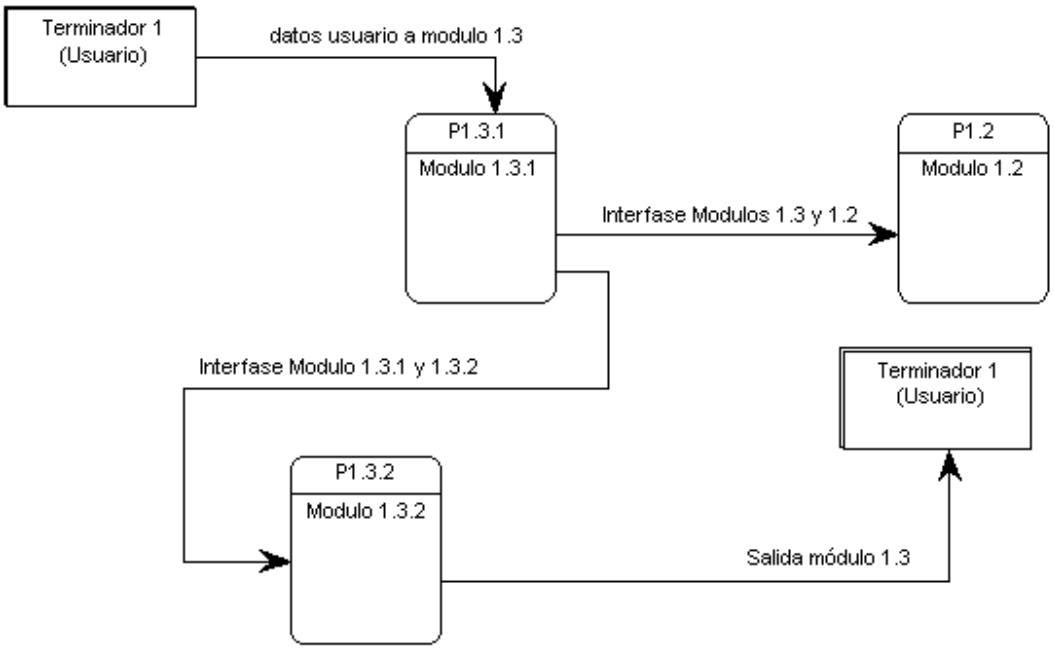


Diagrama nivel 2. En este caso se presenta una descomposición funcional del módulo 1.3



Diccionario de Datos.
La segunda herramienta de modelado importante, aunque no tiene la presencia y atractivo gráfico de los DFD, los diagramas Entidad-Relación o los diagramas de estructuras, es el diccionario de datos.

El diccionario de datos es un listado organizado de todos los datos pertinentes al sistema, con definiciones precisas y rigurosas para que tanto el usuario como el analista tengan un entendimiento común de todas las entradas, salidas, componentes de los almacenes y cálculos intermedios. El diccionario de datos define los datos haciendo lo siguiente:

- Describe el significado de los flujos y almacenes que se muestran en los DFD.

- Describe la composición de agregados de paquetes de datos que se mueven a lo largo de los flujos, es decir, paquetes complejos que pueden descomponerse en unidades más elementales.

- Describen la composición de los paquetes de datos en los almacenes.

- Especifica los valores y unidades relevantes de piezas elementales de información en los flujos de datos y en los almacenes de datos.

- Describe los detalles de las relaciones entre almacenes que se enfatizan en un diagrama de entidad-relación u otro modelo de datos.

Notación del diccionario de datos.
Existen muchos esquemas de notación comunes utilizados. Este es uno de los más utilizados.

= : está compuesto de

+ : y

( ) :optativo (puede estar presente o ausente)

{ } : iteración

[ ] : seleccionar una de varias alternativas

* * : comentario

@ : identificador (campo clave) para un almacén

| : separa opciones alternativas en la construcción

Por ejemplo, podemos definir

nombre = título de cortesía + nombre + (segundo nombre) + apellido paterno + apellido materno

título de cortesía = [Sr. | Srta. | Sra. | Dr. | Profesor]

nombre = {caracter legal}

apellido paterno = {caracter legal}

apellido materno = {caracter legal}

Completitud del Diccionario de Datos.
Para verificar varios detalles de corrección del sistema independientemente del usuario, el analista puede asegurarse que el diccionario esté completo y sea consistente y no contradictorio. Así, puede plantearse las siguientes preguntas:

- ¿Se ha definido en el diccionario cada flujo del DFD?

- ¿Se han definido todos los componentes de los datos en el diccionario?
- ¿Se ha definido más de una vez algún dato?
- ¿Se ha utilizado la notación correcta para todas las definiciones del diccionario de datos?
- ¿Hay elementos de datos en el diccionario que no estén relacionados con el DFD u otros diagramas?

Especificaciones de Proceso.
La especificación del proceso es la descripción de qué es lo que sucede en cada burbuja primitiva en el nivel más bajo en un DFD. También es llamado Minispec o miniespecificación. Su propósito es definir lo que debe hacerse para transformar entradas en salidas.

La forma más utilizada para realizar las especificaciones de procesos es el lenguaje estructurado, pero se puede utilizar cualquier método que satisfaga dos requerimientos cruciales:

- La especificación del proceso debe expresarse de una manera que puedan verificar tanto el usuario como el analista. Precisamente por esta razón se evita el lenguaje narrativo como herramienta de especificación: es ambiguo, sobre todo si describe acciones alternativas y acciones repetitivas. Por naturaleza, también tiende a causar gran confusión cuando expresa condiciones booleanas compuestas (esto es combinaciones de los operadores AND, OR y NOT).

- El proceso debe especificarse en una forma que pueda ser comunicada efectivamente al público amplio que esté involucrado. A pesar de que el analista es típicamente quien escribe la especificación del proceso, habitualmente será un público bastante diverso de usuarios, administradores, auditores, personal de control de calidad y otros, el que leerá la especificación del proceso.

Lenguaje estructurado.
También conocido como español estructurado, es el más utilizado para realizar especificaciones de procesos. Es un subconjunto del español, como lo son del inglés muchos de los lenguajes de programación.

Una frase del lenguaje estructurado puede ser una expresión algebraica:

x= (y*z)/(q+10)

o una frase imperativa consistente de un verbo y un objeto.

Se sugiere seleccionar una cantidad de verbos reducida, como

Conseguir (aceptar, leer)

poner (mostrar, desplegar, escribir)

encontrar (buscar, localizar)

sumar

restar

dividir

multiplicar

calcular

borrar

validar

mover

reemplazar

ordenar

Además se utilizan las estructuras de control de la programación estructurada (if-then-else, while-do, repeat-until, for-do y la concatenación de sentencias) traducidas al español:

Si condición 1 entonces
bloque
sino
bloque alternativo
fin-si

mientras condición hacer
bloque
fin-mientras

repetir
bloque
hasta condición

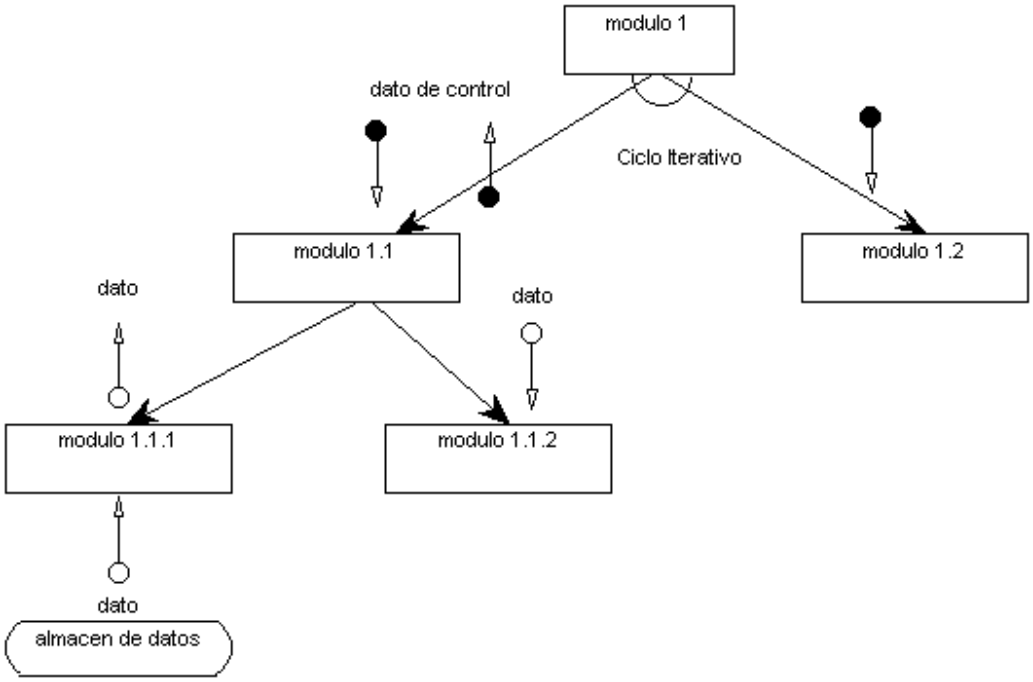
para var desde inicio hasta fin hacer
bloque
fin-para

En general, se pueden hacer adaptaciones a lenguajes como Pascal o SQL para fines de acceso a datos, de modo de utilizarlos en las especificaciones de procesos. También se utilizan tablas de decisión y diagramas de flujo en las minispecs.

Diagramas de Estructura.

A través de los diagramas de estructura se puede modelar el control del sistema, así como la descomposición de las funciones en forma jerárquica.

En un diagrama de estructura, los módulos son representados por rectángulos. Se representa la dependencia (jerárquica) entre módulos, las instancias de repetición y decisión así como el flujo de los datos de control y otros a través de las funciones. Los módulos del diagrama de estructura son los mismos que los que aparecen en los distintos niveles del DFD, vistos en otra dimensión.



Aunque el módulo padre de un diagrama de estructura o módulo raíz puede tener dos o n hijos en su segundo nivel de descomposición, se recomienda descomponer este módulo en 3 hijos, cada uno de ellos dará origen a una rama en el diagrama de estructura, es decir, cada uno de ellos a su vez podrá tener otros módulos hijos.

Estas ramas son:

a. rama aferente: su objetivo es capturar u obtener la información proveniente generalmente del usuario.

b. rama de proceso: transforma la información capturada, es decir las entradas, en las salidas del sistema.

c. rama eferente: su objetivo es entregar las salidas del sistema al usuario o al terminador que corresponda.

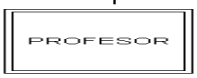
2. Enfoque de datos

- ***Modelo entidad-relación***

Entidades. Se puede definir cono entidad a cualquier objeto, real o abstracto, que existe en un contexto determinado o puede llegar a existir y del cual deseamos guardar información, por ejemplo: "PROFESOR", "CURSO", "ALUMNO". Las entidades las podemos clasificar en:

1. **Regulares:** aquellas que existen por sí mismas y que la existencia de un ejemplar en la entidad no depende de la existencia de otros ejemplares en otra entidad. Por ejemplo "EMPLEADO", "PROFESOR". La representación gráfica dentro del diagrama es la siguiente:

2. **Débiles:** son aquellas entidades en las que se hace necesaria la existencia de ejemplares de otras entidades distintas para que puedan existir ejemplares en esta entidad. Un ejemplo sería la entidad "ALBARÁN" que sólo existe si previamente existe el correspondiente pedido. La representación gráfica dentro del diagrama es la siguiente:



Como complemento al diagrama de entidades del modelo de datos, podemos utilizar la siguiente plantilla para definir las diferentes entidades:

Nombre	PROFESOR
Objeto	Almacenar la información relativa de los profesores de la organización.
Alcance	Se entiende como profesor a aquella persona que, contratada por la organización, imparte, al menos, un curso dentro de la misma.
Número de ejemplares	10 profesores
Crecimiento previsto	2 profesores / año
Confidencialidad	1. Nombre y apellidos: Acceso público. 2. Datos personales: Acceso restringido a secretaría y dirección. 3. Salario: Acceso restringido a dirección.
Derechos de Acceso	Para garantizar la total confidencialidad de esta entidad, el sistema de bases de datos deberá solicitar un usuario y una contraseña para visualizar los elementos de la misma.
Observaciones	Los ejemplares dados de baja no serán eliminados de la base de datos; pasarán a tener una marca de eliminado y no serán visualizados desde la aplicación.

Atributos

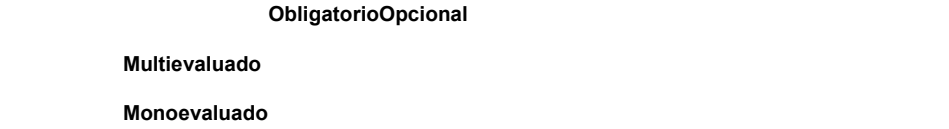
Las entidades se componen de atributos que son cada una de las propiedades o características que tienen las entidades. Cada ejemplar de una misma entidad posee los mismos atributos, tanto en nombre como en número, diferenciándose cada uno de los ejemplares por los valores que toman dichos atributos. Si consideramos la entidad "PROFESOR" y definimos los atributos Nombre, Teléfono y Salario, podríamos obtener los siguientes ejemplares:

{Luis García, 91.555.55.55, 80.500}
{Juan Antonio Alvarez, 91.666.66.66, 92.479}
{Marta López, 91.777.77.77, 85.396}

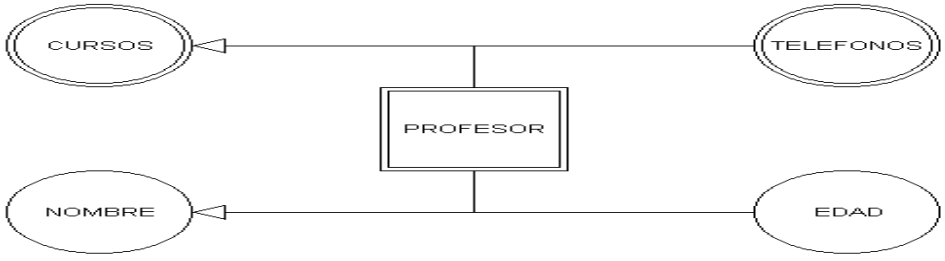
Existen cuatro tipos de atributos:

1. Obligatorios: aquellos que deben tomar un valor y no se permite ningún ejemplar no tenga un valor determinado en el atributo.
2. Opcional: aquellos atributos que pueden tener valores o no tenerlo.
3. Monoevaluado: aquel atributo que sólo puede tener un único valor.
4. Multievaluado: aquellos atributos que pueden tener varios valores.

La representación gráfica de los atributos, en función del tipo es la siguiente:

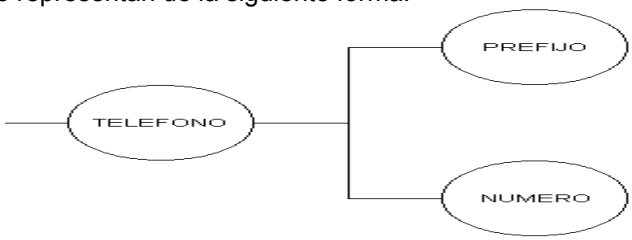


Dentro del diagrama la entidad "PROFESOR" y sus atributos quedaría de la siguiente forma:



Existen atributos, llamados derivados, cuyo valor se obtiene a partir de los valores de otros atributos. Pongamos como ejemplo la entidad "PROFESOR" que tiene los atributos "NOMBRE", "FECHA DE NACIMIENTO", "EDAD"; el atributo "EDAD" es un atributo derivado por que se calcula a partir del valor del atributo "FECHA DE NACIMIENTO". Su representación gráfica es la siguiente:

En determinadas ocasiones es necesaria la descomposición de un atributo para definirlos en más de un dominio, podría ser el caso del atributo "TELÉFONO" que toma valores del dominio "PREFIJOS" y del dominio "NÚMEROS DE TELÉFONO". Estos atributos se representan de la siguiente forma:



Como complemento al diagrama de entidades del modelo de datos, podemos utilizar la siguiente plantilla para definir los diferentes atributos:

Nombre del atributo	**FECHA DE NACIMIENTO**
Tipo de dato	Número largo
Formato interno	aaaammdd
Longitud	8
Formato externo	dd/MM/aaaa
Descripción	Fecha de nacimiento del profesor
Dato requerido	SI
Permitir valor vacío	NO
Valor único	NO
Indexado	SI
Dominio	Calendario Gregoriano
Validaciones	La fecha debe ser superior a 01/01/1900
Confidencial	NO
Derechos de acceso	NO
Observaciones	...

Dominios. Se define dominio como un conjunto de valores que puede tomar un determinado atributo dentro de una entidad. Por ejemplo:

Atributo	**Dominio**
Fecha de Alta	Calendario Gregoriano
Teléfono	Conjunto de números de teléfonos
Cobro de Incentivos	SI / NO
Edad	16 - 65

De forma casi inherente al término dominio aparece el concepto restricción para un atributo. Cada atributo puede adoptar una serie de valores de un dominio restringiendo determinados valores. El atributo "EDAD" toma sus valores del dominio N (números naturales) pero se puede poner como restricción aquellos que estén en el intervalo (0-120), pero dentro de la entidad "PROFESOR" se podría restringir aun más el intervalo, puesto que la edad mínima para trabajar es de 16 años y la máxima de 65, por lo tanto el intervalo sería (16-65).

Claves. El modelo entidad - relación exige que cada entidad tenga un identificador, se trata de un atributo o conjunto de atributos que identifican de forma única a cada uno de los ejemplares de la entidad. De tal forma que ningún par

de ejemplares de la entidad puedan tener el mismo valor en ese identificador. Un ejemplo de identificador es el atributo "DNI" que, en la entidad "ESPAÑOLES", identifica de forma única a cada uno de los españoles. Estos identificadores reciben en nombre de Identificador Principal (IP) o Clave Primaria (PK - Primary Key-). Se puede dar el caso de existir algún identificador más en la entidad, a estos identificadores se les denomina Identificadores Candidatos (IC). Los atributos identificadores de una entidad se representan en los diagramas de la siguiente forma:

Interrelaciones. Se entiende por interrelación a la asociación, vinculación o correspondencia entre entidades. Por ejemplo, entre la entidad "PROFESOR" y la entidad "CURSO" podemos establecer la relación "IMPARTE" por que el profesor imparte cursos. Al igual que las entidades, las interrelaciones se pueden clasificar en regulares y débiles, según estén asociando dos tipos de entidades regulares o una entidad débil con otra de cualquier tipo. Las interrelaciones débiles se subdividen en dos grupos:

1. **En existencia:** cuando los ejemplares de la entidad débil no pueden existir si desaparece el ejemplar de la entidad regular del cual dependen.

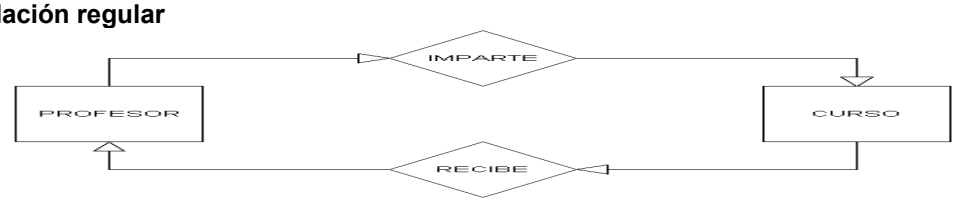
2. **En identificación:** cuando, además de ser una relación en existencia, los ejemplares de la entidad débil no se pueden identificar por sí mismos y exigen añadir el identificador principal de la entidad regular del cual dependen para ser identificados.

Las interrelaciones, dentro de los diagramas, se representan de la siguiente forma:

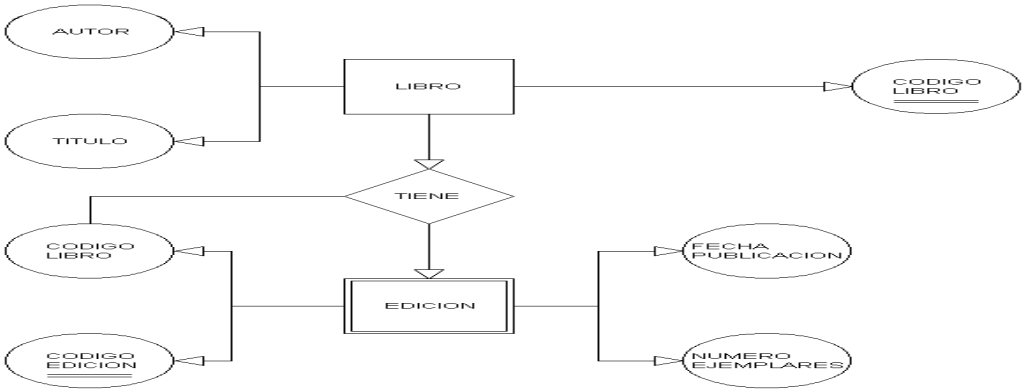
Regulares

Débiles

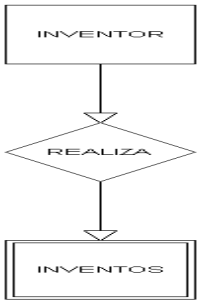
Interrelación regular



Interrelación en identidad

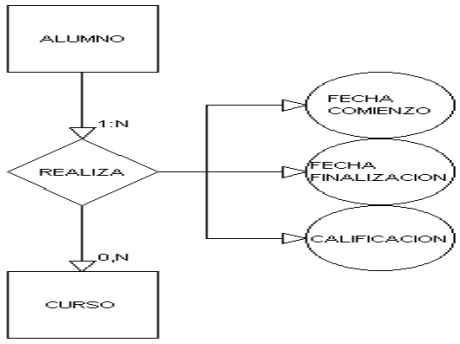


Interrelación en existencia

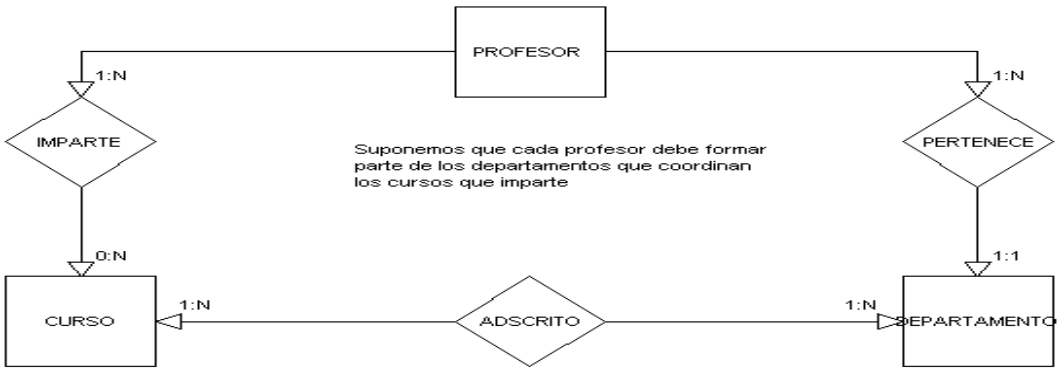


En cada interrelación se debe establecer el número máximo y mínimo de ejemplares de un tipo de entidad que pueden estar asociadas, mediante una determinada relación, con un ejemplar de otra entidad. Este valor máximo y mínimo se conoce como cardinalidad y, según corresponda, se representa de la siguiente forma: (0,n), (n,0), (1,n), (n,1), (0,1), (1,0), (0,0) ó (n,n). La cardinalidad se representa de la siguiente forma:

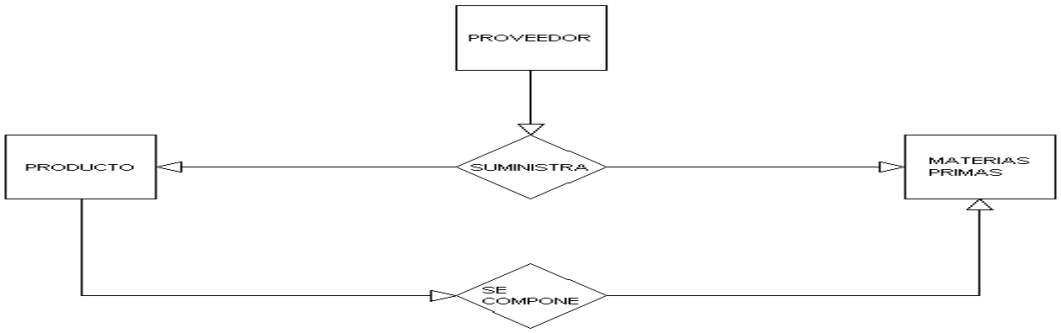
En el diagrama anterior la cardinalidad "CLIENTE" - "PEDIDO" es 1:1 por que al formularnos la pregunta ¿cuántos clientes se pueden relacionar con un pedido? la respuesta es, uno como mínimo y uno como máximo, ya que un pedido es realizado por un único cliente y no cabe la posibilidad que el mismo pedido esté formulado por dos clientes distintos. La cardinalidad "PEDIDO" - "CLIENTE" es 1:N por que la formularnos la pregunta ¿cuántos pedidos se pueden relacionar con un cliente? la respuesta es, como mínimo un pedido pertenece a un cliente, pero varios pedidos pueden estar relacionados con el mismo cliente. Existen ocasiones concretas en que las relaciones tienen atributos, es el caso del diagrama siguiente en donde los alumnos reciben cursos, y la interrelación posee los atributos de fecha de comienzo, fecha de finalización y calificación.



A medida que se van estableciendo las interrelaciones hay que prestar especial atención a las interrelaciones cíclicas o redundantes, que son aquellas que su eliminación no implica la pérdida de información. Pongamos como ejemplo en siguiente modelo entidad - relación:

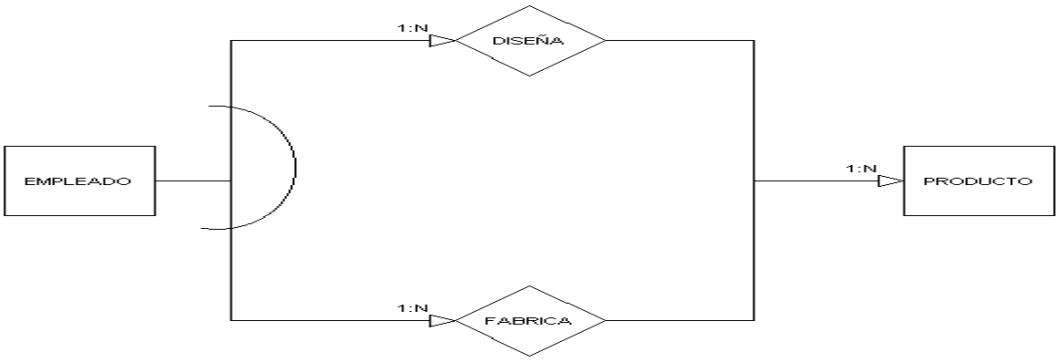


Según se plantea el esquema la relación "PERTENECE" se puede suprimir por que para saber a qué departamentos pertenece un profesor basta con saber que cursos imparte y conociendo los cursos averiguamos que departamentos están asociados a los cursos. En este caso se dice que: "PERTENECE" = "IMPARTE" + "ADSCRITO". En determinadas ocasiones aparecen relaciones que asocian a más de dos entidades, se trata de las relaciones de grado superior. Un ejemplo de este tipo de relación es el siguiente diagrama:



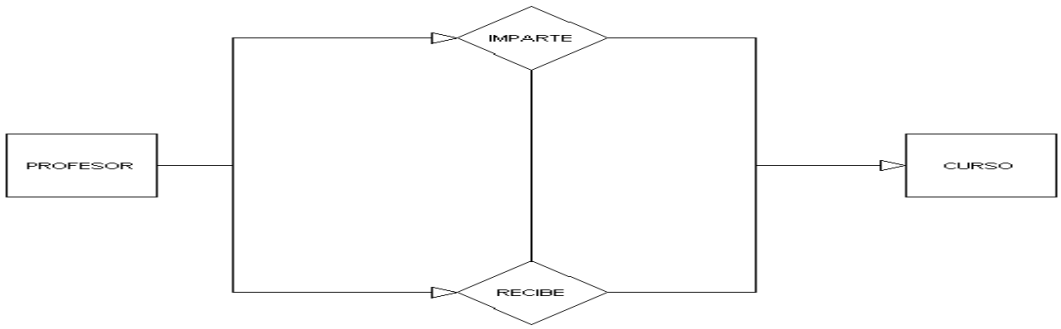
Restricciones en las interrelaciones

Restricción de Exclusividad. Dos o más interrelaciones son de exclusividad cuando cada ejemplar de la entidad presente en todas sólo puede combinarse con ejemplares de una sola de las entidades restantes. Por ejemplo:



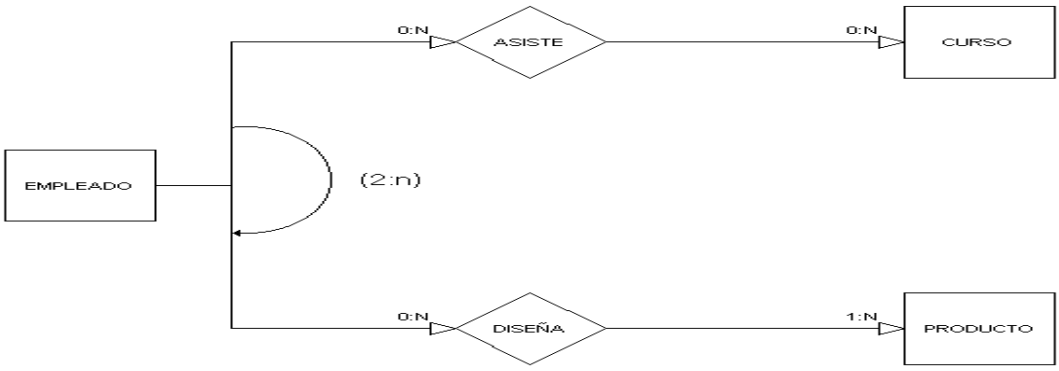
Los empleados, en función de sus capacidades, o son diseñadores de productos o son operarios y los fabrican, no es posible que ningún empleado sea diseñador y fabricante a la misma vez.

Restricción de Exclusión. Se produce una restricción de exclusión cuando los ejemplares de las entidades sólo pueden combinarse utilizando una interrelación. Es el caso del siguiente ejemplo:



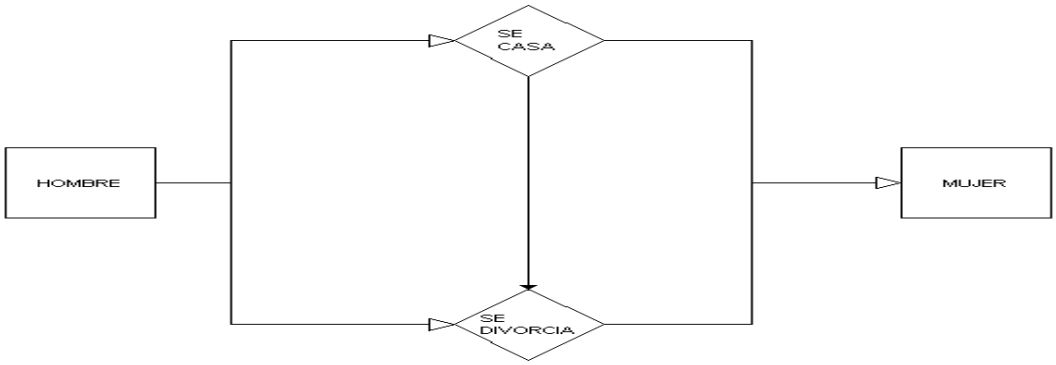
Un profesor no puede recibir e impartir el mismo curso, aunque al contrario que en la restricción anterior puede impartirlo o recibirlo.

Restricción de Inclusividad. Se dice que una relación es de inclusividad cuando todo ejemplar de una entidad que participa en una interrelación ha tenido que participar en la otra. Por ejemplo:



Para que un empleado pueda trabajar como diseñador de productos deber haber asistido, al menos, a dos cursos.

Restricción de Inclusión. Se establece una restricción de inclusión cuando todo ejemplar de una entidad, para participar en la asociación con otro elemento de otra entidad mediante una interrelación, es necesario que ambos elementos estén asociados por una segunda interrelación. Por ejemplo:



Para que un hombre se divorcie de una mujer, previamente ha de haberse casado con ella.

- ***Jackson System Development***

El desarrollo de sistema de Jackson (DSJ) se obtuvo a partir del trabajo de M.A. Jackson sobre el análisis del dominio de la información y sus relaciones con el diseño de programas y sistemas. En palabras de Jackson: "El que desarrolla el software comienza creando un modelo de la realidad a la que se refiere el sistema, la realidad que proporciona su materia objeto [del sistema]..."

Para construir un DSJ el analista aplica los siguientes pasos:

Paso de las acciones y entidades. Usando un método muy similar a la técnica de análisis orientada al objeto, en este paso se identifican las entidades (persona, objetos u organizaciones que necesita un sistema para producir o usar información) y acciones (los sucesos que ocurren en el mudo real que afectan a las entidades).

Paso de estructuración de las entidades. Las acciones que afectan a cada entidad son ordenadas en el tiempo y representadas mediante diagramas de Jackson (una notación similar a un árbol).

Paso de modelación inicial. Las entidades y acciones se representan como un modelo del proceso; se definen las conexiones entre el modelo y el mundo real.

Paso de las funciones. Se especifican las funciones que corresponden alas acciones definidas.

Paso de temporización del sistema. Se establecen y especifican las características de planificación del proceso.

Paso de implementación. Se especifica el hardware y software como un diseño.

Los últimos tres pasos del DSJ están muy relacionados con el diseño de sistemas.

- ***Metodología Warnier-Orr***

Diagramas de Warnier/Orr

Los diagramas de Warnier/Orr son un tipo de diagramas jerárquicos que se utilizan para describir tanto la organización de datos como de procedimientos. Hay cuatro construcciones básicas utilizadas en los diagramas de W/O: jerarquía, secuencia, repetición, y selección. Tambien hay dos conceptos avanzados que ocacionalmente son necesarios: concurrencia y recursión.

Jerarquía

La jerarquía es la construcción mas importante. Consiste simplemente en un grupo anidado de conjutnos y subconjuntos representados por un conjunto de llaves anidadas. El siguiente es un ejemplo de una jerarquía sencilla:
Cada llave en el diagrama representa un nivel de la jerarquía. El diagrama puede ser utilizado para representar una jerarquía de datos, o de procedimiento. Cada llave puede ser interpretada con la frase “consiste de” o “ está compuesto por”.

Secuencia

La secuencia es la estructura más simple de un diagrama W/O. Dentro de un nivel de una
jerarquía, las características listadas son presentadas en el orden en que ocurren.

Repetición

La repetición es la representación del clásico loop en términos de programación. Para una estructura de datos significa que el mismo conjunto de datos se repite muchas veces. Para una estructura de proceso significa que el mismo conjunto de acciones se repiten muchas veces. La repetición es indicada colocando un para ordenado de números entre paréntesis debajo del conjunto repetitivo. Típicamente el par de números representa el mínimo y máximo número de veces que ocurre la repetición, aunque puede representar valores exactos.

Selección

La selección representa una decisión u or exclusivo entre los conjuntos implicados.

Concurrencia

Es usado siempre que la secuencia no es importante, y ocurren ambas cosas.

Recursión

La recursión es la menos usada de las construcciones. Se utiliza cuando un conjunto contiene una versión de si mismo. Se simboliza con una doble llave.

Desarrollo de Sistemas Estructurados en Datos DSED (metodología Warnier-Orr)

A partir de las especificaciones de requisitos se realiza el diseño lógico y físico. El diseño lógico se centra en las salidas, en las interfaces, y en el diseño procedimental del software. El diseño físico surge del diseño lógico y se centra en el “empaquetamiento” del sw para conseguir lo mejor posible el rendimiento deseado, la facilidad de mantenimiento, y otras restricciones impuestas al diseño. El proceso de diseño lógico puede dividirse en dos actividades:

· Derivación de la estructura lógica de la salida (ELS)
· Derivación de la estructura lógica del proceso (ELP)

Derivación de la ELS

1. Se evalúa la descripción del problema o la información relativa a los requisitos y se listan todos los elementos de datos diferentes, denominados átomos, que no puedan subdividirse más.
2. Se especifica la frecuencia de ocurrencia de cada átomo.
3. Se evalúan los elementos de datos compuestos llamados universales. Los universales son elementos de datos que están compuestos de otros universales y átomos.
4. Se desarrolla la representación diagramática de la ELS.

Derivación de la ELP

1. Se quitan todos los átomos del diagrama de Warnier.
2. Se añaden los delimitadores BEGIN y END a todos los universales (repeticiones)
3. Se definen todas las instrucciones o procesos de inicialización y terminación (los begin-end).
4. Se especifican todos los cálculos o procesamientos no numéricos.
5. Se especifican todas las instrucciones y procesos de salida.
6. Se especifican todas las instrucciones y procesos de entrada.

3. Enfoque orientado a objetos

- ***Conceptos generales (características y conceptos del modelo de objetos, clases y objetos, relaciones entre objetos relaciones entre clases)***

Teoría de objeto: Para entender la idea fundamental que subyace la tecnología orientada a objetos comenzaremos por definir lo que es un objeto:
¿Que es un objeto? El termino objeto surgió casi independientemente en varios campos de la informática, simultáneamente a principios de los sesenta, para referirse a nociones que eran diferentes en su apariencia, pero relacionados entre sí. Todas estas nociones se inventaron para manejar la complejidad de sistemas software de tal forma que los objetos representaban componentes de un sistema descompuesto modularmente o bien unidades modulares de representación del conocimiento. Desde temprana edad las personas nos formamos conceptos, cada concepto es una idea particular o una comprensión de nuestro mundo, los conceptos adquiridos nos permiten sentir y razonar a cerca de las cosas en el mundo. A estas cosas a las que se aplican nuestros conceptos se llaman objetos. Un objeto puede ser real o abstracto, como:

- Una factura
- Una organización
- Una figura en un programa
- Un avión
- Etc.

"Un objeto es cualquier cosa, real o abstracta, a cerca de la cual almacenamos datos y los métodos que controlan dichos datos."
Un objeto puede estar compuesto por otros objetos, estos últimos a su vez, pueden estar compuestos de objetos, del mismo modo que una maquina esta formada por partes y estas, también, están formadas por otras partes. Esta estructura intrincada de los objetos permite definir objetos muy complejos. Los objetos tienen una cierta integridad que no debería -de echo no puede- ser violada. Un objeto solo puede cambiar de estado, actuar, ser manipulado o permanecer en relación con otros objetos de maneras apropiadas para ser objeto. Las técnicas orientadas objetos permiten que el software se construya a partir de objetos de comportamiento especifico. Como lo mencionamos antes, los objetos se pueden construir a partir de otros, que a su vez pueden estar formados por otros objetos. Esto nos recuerda una maquinaria compleja, construida por partes, subpartes, sub-subpartes, etc. El análisis de sistemas en el mundo orientado a objetos se realiza al estudiar los objetos en un ambiente.
¿Qué es un tipo de objeto? Los conceptos que poseemos se aplican a tipos determinados de objetos.
Un tipo de objeto es una categoría de objeto. Un objeto es una instancia de un tipo de objeto
Por ejemplo, empleado se aplica a los objetos que son personas empleadas por alguna organización. Algunas instancias de empleado podrían ser Juan Pérez, María Sánchez, Etc. En el análisis orientado a objetos, estos conceptos se llaman tipos de objetos. Otro ejemplo, es que un tipo de objeto podría ser factura y un objeto podría ser Factura Nº 51783. Sin embargo, él termino objeto tiene diferencias fundamentales con él termino ente. Ente solo se refiere a los datos. Objeto se refiera a los datos y los métodos mediante los cuales se controlan a los propios datos. En el mundo OO, la estructura de datos y los métodos de cada tipo de objeto se manejan juntos. No se puede tener acceso o control de la estructura de datos excepto mediante los métodos que forman parte del tipo de objeto. Por ejemplo actividades como contratar, promover, retirar y despedir están ligadas íntimamente con el tipo de objeto empleado, puesto que cambian su estado. En otras palabras un objeto solo debe ser controlado por medio de las funciones asociadas con su tipo. Así, las operaciones no se pueden definir de manera adecuada sin los tipos de objetos.
Elementos fundamentales del análisis orientado a objetos
El modelo de objetos
El modelo de objetos describe la estructura de los objetos de un sistema: su identidad, sus relaciones con otros objetos, sus atributos y sus operaciones. Los cambios y las transformaciones no tienen sentido a menos que haya algo que cambiar o transformar. El modelo de objetos se representa gráficamente con diagramas de objetos y diagramas de instancias, que contienen clases de objetos e instancias respectivamente. Las clases se disponen en jerarquías que compartan una estructura de datos y un comportamiento común, y se relacionan con otras clases. Cada clase define los atributos que contiene cada uno de los objetos o instancias y las operaciones que realizan o sufren. La tecnología orientada a objetos se apoya en sólidos fundamentos de la ingeniería, cuyos elementos reciben el nombre global de modelo de objetos. El modelo de objetos abarca principios fundamentales que los detallaremos mas adelante. Quede claro que el diseño orientado a objetos es fundamentalmente diferente a los enfoques de diseño estructurado tradicionales; requiere un modo distinto de pensar acerca de la descomposición y produce arquitecturas software muy alejadas del dominio de la cultura del diseño estructurado. Además, conviene mencionar que se observa que la mayoría de los programadores trabajan en un lenguaje y utilizan solo un estilo de

programación. Programan bajo un paradigma apoyado por el lenguaje que usan. Frecuentemente, no se les han mostrado vías alternativas para pensar sobre un problema, y por lo tanto tiene dificultades para apreciar las ventajas de elegir un estilo, mas apropiado para el problema que tienen entre manos. No hay un estilo de programación que sea el mejor para todo tipo de aplicaciones. Por ejemplo, la programación orientada a reglas seria la mejor para el diseño de una base de conocimiento, y la programación orientada a procedimientos seria la mas indicada parea el diseño de operaciones de calculo intensivo. Por experiencia de algunos estudiosos de la materia, el estilo orientado a objetos es él mas adecuado para él más amplio conjunto de aplicaciones; realmente, este paradigma de programación sirve con frecuencia como el marco de referencia arquitectónico en el que se emplean otros paradigmas. Cada uno de estos estilos de programación se basa en su propio marco de referencia conceptual. Cada uno requiere una actitud mental diferente, una forma distinta de pensar en el problema. Para todas las cosas orientadas a objetos, el marco de referencia conceptual es el modelo de objetos.

Fundamentos del modelo de objetos:

En realidad, el modelo de objetos ha recibido la influencia de una serie de factores, no solo de la programación orientada a objetos. El modelo de objetos a demostrado ser un concepto u unificador de en la informática. La razón de este gran atractivo es simplemente que una orientación a objetos ayuda a combatir la complejidad inherente a muchos tipos de sistemas diferentes. El diseño orientado a objetos representa así un desarrollo evolutivo, no revolucionario; no rompe con los avances del pasado, sino que se basa en avances ya probados. Los métodos de diseñó estructurado surgieron para guiar a los desarrolladores que intentaban construir sistemas complejos utilizando los algoritmos como bloques fundamentales para su construcción. Análogamente los métodos de diseño orientados a objetos han surgido para ayudar a los desarrolladores a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases como bloques básicos de construcción. Desgraciadamente, hoy en día la mayoría de los programadores han sido educados formal e informalmente solo en los principios del diseño estructurado. En el modelo de objetos es necesario estudiar los principios fundamentales en los que se basa el análisis orientado a objetos, es decir;

- Abstracción
- Encapsulación
- Modularidad
- Jerarquía
- Concurrencia

Ninguno de estos principios es nuevo por sí mismo. Lo importante del modelo de objetos es el hecho de conjugar todos estos elementos en forma sinérgica. Al decir fundamentales, quiere decir que un modelo que carezca de cualquiera de estos elementos no estará orientado a objetos. Otros elementos que podrían llamarse secundarios, que quiere decir, que cada uno de ellos es una parte útil del modelo de objetos, pero no esenciales son:

- Tipos
- Persistencia

A partir de los elementos antes mencionados, trataremos de mostrar, mediante una definición en primer lugar de cada uno de ellos, cual es la asociación con el sistema solar y el sistema celular, esto quiere decir que asociaremos estos elementos bases de la llamada teoría de los objetos con el macrocosmo y el microcosmo, dictando sus características fundamentales para el entendimiento claro de este. A la vez, también definiremos las desigualdades u oposiciones de estos elementos con los sistemas antes mencionados. En otras palabras veremos como pasamos de un análisis estructurado a un análisis orientado a objetos, y viceversa. Macrocosmo v/s Microcosmo, esto incorporado al sistema de la teoría de los objetos.

Conceptos bases y elementales: Las técnicas orientadas a objetos se basan en organizar el software como una colección de objetos discretos que incorporan tanto estructuras de datos como comportamiento. Esto contrasta con la programación convencional, en la que las estructuras de datos y el comportamiento estaban escasamente relacionadas. Las características principales del enfoque orientado a objetos son, en primer lugar:

- Objeto
- Clase
- Método
- Polimorfismo
- Encapsulación
- Herencia
- Mensajes
- Identidad
- Reusabilidad
- Jerarquía

- Abstracción
- Concurrencia
- Modularidad

A partir de estos elementos fundamentales, trataremos de dar un enfoque tanto estructurado como también un enfoque orientado a objetos, principal motivo de nuestro trabajo de investigación encomendado, debemos destacar que los elementos antes mencionados son un conjunto de información y bases que se investigo de libros o manuales alusivos al tema tratado, que de forma de organizar y abarcar distintas opiniones de especialistas en esta rama, formamos un todo para explicar y dar un buen entendimiento a la teoría del análisis orientado a los objetos. Para comenzar, daremos un enfoque básico y entendible como también, singular del material investigado, que es la definición según enciclopedias y diccionarios que nada tienen que ver con el análisis directamente orientado a objetos, esto con motivo de hacerlo practico y de modo que el lector de esto se pueda familiarizar con estos componentes, básicos en nuestro trabajo de investigación, para luego y de una forma mas complicada o a la vez, mas asociada al trabajo en si, podamos asociarlos y compararlos con el mundo de la computación netamente orientado a objetos, como también con al sistema celular y solar.

Definiciones basicas:

OBJETO : Fin, intento, propósito. Materia y sujeto de una ciencia.
CLASE : Orden de cosas de una misma especie. Conjunto de ordenes.
POLIMORFISMO : Propiedad de los cuerpos que cambian de forma sin cambiar su naturaleza. Presencia de distintas formas individuales en una sola especie.
ENCAPSULACION : Proceso de constitución de una cápsula.
HERENCIA : Derecho de suceder a otro la posesión de bienes o acciones.
MENSAJE : Información que se le envía a alguien.
METODO : Modo de hacer en orden una cosa, modo habitual de procede.
IDENTIDAD : Cualidad de ser lo mismo que otra cosa con que se compara.
REUSABILIDAD : Acción de volver a utilizar. Que puede volver a ser utilizado.
JERARQUIA : Orden o grados de una especie.
ABSTRACCION : Separación, apartamiento, aislamiento, prescindir.
CONCURRENCIA : Asistencia, reunión simultanea de personas o cosas.
MODULARIDAD : Acción de pasar de un termino a otro.

- ***Análisis orientado a objetos (el modelo de objetos, el modelo dinámico, el modelo funcional)***

Análisis estructurado v/s análisis orientado a objetos

Análisis estructurado de objetos

Esta es una alternativa para que el análisis orientado a objetos clásico, utilice los productos del análisis estructurado como vía de entrada al diseño orientado a objetos. En esta aproximación, se comienza con un modelo esencial del sistema, tal como lo describen los diagramas de flujo de datos y otros productos del análisis estructurado. El análisis de la estructura de objetos (AEO) define las categorías de los objetos que percibimos y las formas en que los asociamos. Durante el análisis de las estructuras de objetos, el equipo de análisis se preocupa mas por identificar los tipos de objetos que por identificar los objetos individuales en unos sistemas. En el análisis de la estructura de objetos se identifica lo siguiente:

-¿Qué son los tipos de objetos y como se asocian?

La identificación de los objetos y sus asociaciones se representan mediante esquemas de objetos. Esta información guía al diseñador en la definición de clases y estructuras de datos.

-¿Cómo se organizan los tipos de objetos en superfinos y subtipos?

La jerarquía de generalizaciones pueden organizar en diagramas e indicar al diseñador las direcciones de herencia.

-¿Cuál es la composición de los objetos complejos?

Se pueden elaborar diagramas de jerarquías compuestas. La composición guía al diseñador en la definición de mecanismos que controlen adecuadamente a los objetos dentro de otros objetos.

Cuando el análisis pasa a la etapa del diseño de la estructura de objetos, identificamos las clases. Se definen las superclases, subclases, rutas de herencia y los métodos a utilizar y se lleva a cabo el detalle del diseño de la estructura de datos.

Análisis orientado a objetos

La experiencia de algunos analistas nos lleva a aplicar en primer lugar el criterio orientado a objetos porque esta aproximación es mejor a la hora de servir de ayuda para organizar la complejidad innata de los sistemas de software, al igual que ha servido de ayuda para describir la complejidad organizada de sistemas complejos tan diversos como

los computadores, plantas, galaxias o grandes instituciones sociales. Los sistemas orientados a objetos son también más resistentes al cambio y por lo tanto están mejor preparados para evolucionar en el tiempo, porque su diseño esta basado en formas intermedias estables. El modelo de objetos ha influido incluso en las fases iniciales del ciclo de vida del desarrollo del software. El análisis orientado a objetos (AOO) enfatiza la construcción de modelos del mundo real utilizando una visión del mundo orientado a objetos: El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema. Básicamente los productos del análisis orientado a objetos sirven como modelos de los que se puede partir para un diseño orientado a objetos; los productos del diseño orientado a objetos pueden utilizarse entonces como anteproyectos para la implementación completa de unos sistemas utilizando métodos de programación orientado a objetos, de esta forma se relacionan AOO, DOO y POO. Se insiste que se ha encontrado un gran valor en la construcción de modelos que se centran en las "cosas" que se encuentran en el espacio del problema formando lo que se ha llamado una descomposición orientada a objetos. El diseño orientado a objetos es el método que lleva a una descomposición orientado a objetos. Ofrece un rico conjuntos de modelos que reflejan la importancia de plasmar explícitamente las jerarquías de clases y de objetos de los sistemas que diseña. El análisis orientado a objetos (AOO) se basa en conceptos sencillos, conocidos desde la infancia y que aplicamos continuamente: objetos y atributos, él todo y las partes, clases y miembros. Puede parecer llamativo que se haya tardado tanto tiempo en aplicar estos conceptos al desarrollo de software. Posiblemente, una de las razones es el éxito de los métodos de análisis estructurados, basados en los conceptos de flujo de información, que monopolizaron el análisis de sistemas de software durante los últimos veinte años. El AOO ofrece un enfoque nuevo para el análisis de requisitos de sistemas software. En lugar de considerar el software desde una perspectiva clásica de entrada - proceso - salida, como los métodos estructurados clásicos se basan en modelar el sistema mediante los objetos que forman parte de el y las relaciones estáticas o dinámicas entre estos objetos. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real a partir del conocimiento del llamado dominio del problema.

Un punto de vista:

Desde el punto de vista de los análisis antes mencionados hablamos del AOO y el AEO, podemos decir que el AOO concibe una abstracción mayor que el AEO, que modela los sistemas desde un punto de vista más próximo a su implementaron en un ordenador (entrada/proceso/salida). La ventaja del AOO es que se basa en la utilización de objetos como abstracciones del mundo real. Esto nos permite centrarnos en los aspectos significativos del domino del problema (en las características de los objetos y las relaciones que se establecen entre ellos) y este conocimiento se convierte en la parte fundamental del análisis del sistema software que será utilizado luego en el diseño y la implementación. En el AOO los objetos encapsulan tanto atributos como procedimientos (operaciones que se realizan sobre los objetos), e incorpora, además, conceptos como los polimorfismos o la herencia que facilitan la reutilización de código. Podemos concluir entonces que el AOO puede facilitar mucho la creación de prototipos, y las técnicas de desarrollo evolutivo de software. Los objetos don inherentemente reutilizables, y se puede crear un catalogo de objetos que podemos usar en sucesivas aplicaciones. De esta forma, podemos obtener rápidamente un prototipo del sistema, que pueda ser evaluado por el cliente, a partir de los objetos analizados, diseñados e implementados en aplicaciones anteriores. Y lo que es más importante, dada la facilidad de reutilización de estos objetos, el prototipo puede ir evolucionando hacia convertirse en el sistema final, según vamos refinando los objetos de acuerdo a un proceso de especificación incremental.

Ventajas del AOO:

Dominio del problema

El paradigma OO es mas que una forma de programar. Es una forma de pensar acerca de un problema desde el punto de vista del mundo real en vez de desde el punto de vista del ordenador. El AOO permite analizar mejor el dominio del problema, sin pensar en términos de implementar el sistema de un ordenador, permite, además, pasar directamente el dominio del problema al modelo del sistema.

Comunicación

El concepto OO es más simple y esta menos relacionado con la informática que el concepto de flujo de datos. Esto permite una mejor comunicación entre el analista y el experto en el dominio del problema.

Consistencia

Los objetos encapsulan tanto atributos como operaciones. Debido a esto, el AOO reduce la distancia entre el punto de vista de los datos y el punto de vista del proceso, dejando menos lugar a inconsistencias disparidades entre ambos modelos.

Expresión de características comunes

El paradigma lo utiliza la herencia para expresar explícitamente las características comunes de una serie de objetos estas característica comunes quedan escondidas en otros enfoques y llevan a duplicar entidades en el análisis y

código en los programas. Sin embargo, el paradigma OO pone especial énfasis en la reutilización y proporciona mecanismos efectivos que permiten reutilizar aquello que es común sin impedir por ello describir las diferencias.
Resistencia al cambio
Los cambios en los requisitos afectan notablemente a la funcionalidad de un sistema por lo que afectan mucho al software desarrollando con métodos estructurados. Sin embargo, los cambios afectan en mucha menos medida a los objetos que componen o maneja el sistema, que son mucho más estables. Las modificaciones necesarias para adaptar una aplicación basada en objetos a un cambio de requisitos suelen estar mucho más localizadas.
Reutilización
Aparte de la reutilización interna, basada en la expresión explícita de características comunes, el paradigma 00 desarrolla modelos mucho más próximos al mundo real, con lo que aumentan las posibilidades de reutilización. Es probable que en futuras aplicaciones nos encontremos con objetos iguales o similares a los de la actual.

Desventajas del AOO

Descomposición funcional
El análisis estructurado se basa fundamentalmente en la descomposición funcional del sistema que queremos construir. Esta descomposición funcional requiere traducir el dominio del problema en una serie de funciones y subfunciones, es decir, el analista debe comprender primero el dominio del problema y a continuación documentar las funciones y subfunciones que debe proporcionar el sistema. El problema es que no existe un mecanismo para comprobar si la especificación del sistema expresa con exactitud los requisitos del sistema.
Flujo de datos
El análisis estructurado muestra como fluye la información a través del sistema. Aunque este enfoque se adapta bien al uso de sistemas informáticos para implementar al sistema, no es la forma habitual de pensar.
Modelo de datos
El análisis estructurado moderno incorpora modelos de datos, además de modelos de procesos y de comportamiento. Sin embargo, la relación entre los modelos es muy débil, y hay muy poca influencia de un modelo en otro. En la practica, los modelos de procesos y de datos de un mismo sistema se parecen muy poco. En muchos casos son visiones irreconciliables, no del mismo sistema sino de dos puntos de vista totalmente diferentes de organizar la solución. Lo ideal seria que ambos modelos se complementasen, no por oposición si no de forma que el desarrollo de uno facilitase el desarrollo del otro.

- ***Diseño orientado a objetos (definición de la arquitectura del sistema, diseño del componente del dominio: de la interfase humana, del problema, de la administración de tareas y de la administración de datos, refinamiento de clases y objetos)***

La esencia del desarrollo de software OO es la identificación y organización de conceptos del dominio del problema, más que en su implementación final usando un determinado lenguaje.

La Técnica de Modelado de Objetos (OMT, Rumbaugh, 1991) es un procedimiento que se basa en aplicar el enfoque orientado a objetos a todo el proceso de desarrollo de un sistema software, desde el análisis hasta la implementación. Los métodos de análisis y diseño que propone son independientes del lenguaje de programación que se emplee para la implementación. Incluso esta implementación no tiene que basarse necesariamente en un lenguaje OO.

OMT es una metodología OO de desarrollo de software basada en una notación gráfica para representar conceptos OO. La metodología consiste en construir un modelo del dominio de aplicación y ir añadiendo detalles a este modelo durante la fase de diseño. OMT consta de las siguientes fases o etapas.

Fases.

- **Conceptualización**. Consiste en la primera aproximación al problema que se debe resolver. Se realiza una lista inicial de requisitos y se describen los casos de uso.

- **Análisis**. El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.

- **Diseño del sistema**. El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la arquitectura propuesta.

- **Diseño de objetos**. El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y algoritmos que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, bases de datos, etc.).

- **Implementación**. Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos AOO y DOO de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

Algunas clases que aparecen en el sistema final no son parte del análisis sino que se introducen durante el diseño o la implementación. Este es el caso de estructuras como árboles, listas enlazadas o tablas *hash*, que no suelen estar presentes en el dominio de aplicación. Estas clases se añaden para permitir utilizar determinados algoritmos.

Los conceptos del paradigma OO pueden aplicarse durante todo el ciclo de desarrollo del software, desde el análisis a la implementación sin cambios de notación, sólo añadiendo progresivamente detalles al modelo inicial.

Modelos.

Al igual que los métodos estructurados, OMT utiliza tres tipos de modelos para describir un sistema:

- **Modelo de objetos**. Describe la estructura estática de los objetos de un sistema y sus relaciones. El modelo de objetos contiene diagramas de objetos. Un diagrama de objetos es un grafo cuyos nodos son clases y cuyos arcos son relaciones entre clases.

- **Modelo dinámico**. El modelo dinámico describe las características de un sistema que cambia a lo largo del tiempo. Se utiliza para especificar los aspectos de control de un sistema. Para representarlo utilizaremos DEs.

- **Modelo funcional**. Describe las transformaciones de datos del sistema. El modelo funcional contiene DFDs y especificaciones de proceso.

Los tres modelos son vistas ortogonales (independientes) del mismo sistema, aunque existen relaciones entre ellos. Cada modelo contiene referencias a elementos de los otros dos. Por ejemplo, las operaciones que se asocian a los objetos del modelo de objetos figuran también, de forma más detallada en el modelo funcional. El más importante de los tres es el modelo de objetos, porque es necesario describir qué cambia antes que decir cuándo o cómo cambia.

Diferencias con los métodos estructurados.

OMT invierte el método estructurado, en el que se da más importancia a la descomposición funcional del sistema y, por tanto, a los diagramas de proceso. Este enfoque de descomposición funcional puede parecer que lleva de forma más directa a una implementación del sistema, pero con frecuencia este sistema suele ser más frágil. Si cambian los requisitos, un sistema basado en descomposición funcional puede requerir una reestructuración masiva. Por el contrario, el enfoque OO se centra en primer lugar en identificar los objetos del dominio de aplicación y después en establecer procedimientos que los manejen. Aunque esto puede parecer más indirecto, el software OO se mantiene mejor ante los cambios de requisitos, porque se basa en la estructura subyacente del dominio de aplicación, en vez de en los requisitos funcionales de un determinado problema.

De todas formas, los modelos son los mismos que en el análisis estructurado. El modelo de objetos, el único que parece nuevo, es bastante similar a los DERs. Las diferencias principales consisten en la mayor importancia que se

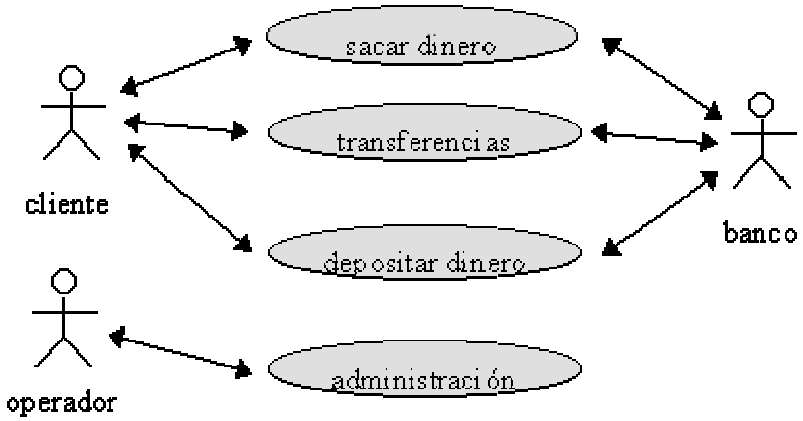
da el modelo de datos, por encima de los otros dos, y en el enfoque orientado a objetos de este modelo (encapsulando tanto datos como operaciones) en vez de estar orientado a bases de datos como los DERs.

En este sentido, puede compararse a los paquetes de lenguajes como ADA, con los que se implementan TADs. La diferencia reside en que en la POO, la herencia y el polimorfismo favorecen la reutilización de código y la programación por especialización, desde una perspectiva distinta al uso de paquetes genéricos en ADA.

Casos de uso.

Una forma de describir los requisitos iniciales del usuario, durante la fase de conceptualización, es construir casos de uso del sistema, descritos inicialmente por Jacobson en 1987 y actualmente incorporados a la mayor parte de las metodologías de AOO.

Un caso de uso está formado por una serie de interacciones entre el sistema y un *actor* (una entidad externa, ejerciendo un rol determinado), que muestran una determinada forma de utilizar el sistema. Cada interacción comienza con un evento inicial que el actor envía al sistema y continua con una serie de eventos entre el actor, el sistema y posiblemente otros actores involucrados.



Un caso de uso puede ser descrito en lenguaje natural, mediante trazas de eventos o mediante diagramas de interacción de objetos.

Modelo de objetos.

El modelo de objetos describe la estructura de los objetos de un sistema: su identidad, sus relaciones con otros objetos, sus atributos y sus operaciones. Los cambios y las transformaciones no tienen sentido a menos que haya algo que cambiar o transformar. OMT considera este modelo el más importante de los tres.

El modelo de objetos se representa gráficamente con diagramas de objetos y diagramas de instancias, que contienen clases de objetos e instancias, respectivamente. Las clases se disponen en jerarquías que comparten una estructura de datos y un comportamiento comunes, y se relacionan con otras clases. Cada clase define los atributos que contiene cada uno de los objetos o instancias y las operaciones que realizan o sufren estos objetos.

Elementos del modelo de objetos.

El modelo de objetos puede contener los siguientes elementos:

Objetos o instancias.

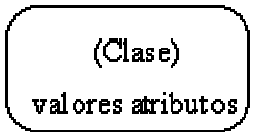
Un objeto es un concepto, una abstracción o una cosa con unos límites definidos y que es relevante para el problema en cuestión. Los modelos de objetos sirven tanto para obtener un conocimiento mejor del dominio de aplicación como de base para la implementación del sistema en un ordenador.

Una característica de los objetos es que tienen identidad y son distinguibles. Aunque dos objetos tengan los mismos valores para todos sus atributos son diferentes.

El término objeto está sobrecargado. Mediante él podemos referirnos tanto a clases de objetos (p. ej. el concepto abstracto mesa) como a las instancias de estas clases (una mesa determinada). Es mejor utilizar los términos clase e instancia para evitar confusiones.

La mayoría de las instancias de una clase derivan su individualidad de tener valores diferentes en alguno/s de sus atributos o de tener relaciones con instancias diferentes. No obstante pueden existir instancias con los mismos valores de los atributos e idénticas relaciones.

El símbolo gráfico para representar instancias es un rectángulo de esquinas redondeadas. Dentro del rectángulo figura la clase a la que pertenece la instancia (entre paréntesis) y los valores de sus atributos.



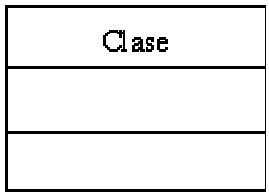
Las instancias figuran en diagramas de instancias, que se utilizan normalmente para describir ejemplos que aclaren un diagrama de objetos complejos o para describir escenarios determinados (p. ej. situaciones típicas o anómalas, escenarios de prueba, etc.).

Clases.

Una clase o clase de objetos es una abstracción que describe un grupo de instancias con propiedades (atributos) comunes, comportamiento (operaciones) común, relaciones comunes con otros objetos y (lo que es más importante) **una semántica común**. Así un *caballo* y un *establo* tienen los dos un *coste* y una *edad*, pero posiblemente pertenezcan a clases diferentes (aunque esto depende del dominio de aplicación: en una aplicación financiera ambos pertenecerían posiblemente a la misma clase: *Inversiones*).

La diferencia entre instancia y clase está en el grado de abstracción. Un objeto es una abstracción de un objeto del mundo real, pero una clase es una abstracción de un grupo de objetos del mundo real. La abstracción permite la generalización y evita la redefinición de las características (atributos, comportamiento o relaciones) comunes, de forma que se produce una reutilización de estas definiciones comunes por parte de cada uno de los objetos. Por ejemplo todas las elipses (instancias) comparten las mismos operaciones para dibujarlas o calcular su área.

El símbolo gráfico para representar clases es un rectángulo, en el que figura el nombre de la clase. Las clases se representan en los diagramas de clases, que son plantillas que describen un conjunto de posibles diagramas de instancias. Describen, por tanto el caso general.

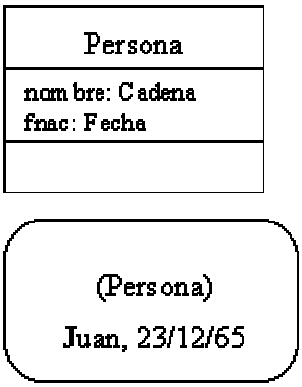


Atributos.

Un atributo es un dato contenido en todas las instancias de una clase. Cada atributo tiene un valor para cada una de las instancias. Varias clases pueden tener atributos comunes (p. ej. *nombre*, en las clases *Persona* y *Calle*) pero cada atributo debe ser único dentro de una clase.

Los atributos tienen que ser datos, no objetos. La diferencia entre unos y otros reside en la identidad: los objetos tienen identidad, pero los atributos no. Por ejemplo, todas las ocurrencias del valor '3' de un atributo son indistinguibles.

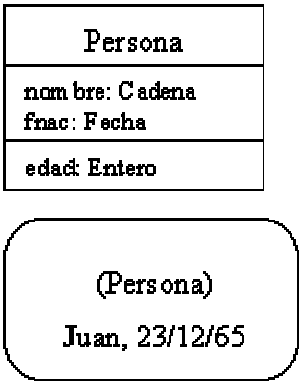
Los atributos se representan en el segundo área de los símbolos de clase e instancia. En las clases, figurará el nombre del atributo, el tipo y el valor por defecto. En las instancias, el valor del atributo para ese objeto determinado.



Operaciones.

Una operación o método es una función o transformación. Cada operación lleva implícito un objeto destino, sobre el que se va a realizar la operación. El comportamiento de la operación depende de la clase del objeto destino. Todos los objetos de una clase comparten las mismas operaciones o métodos. Cada objeto conoce la clase a que pertenece y, por tanto, la implementación correcta de la operación. Una misma operación puede aplicarse a objetos de clases distintas. En este caso diremos que la operación es polimórfica, y a la implementación de la operación en cada una de las clases la llamaremos método.

Una operación puede tener una serie de argumentos explícitos, además del objeto destino, que actúa siempre como argumento implícito.



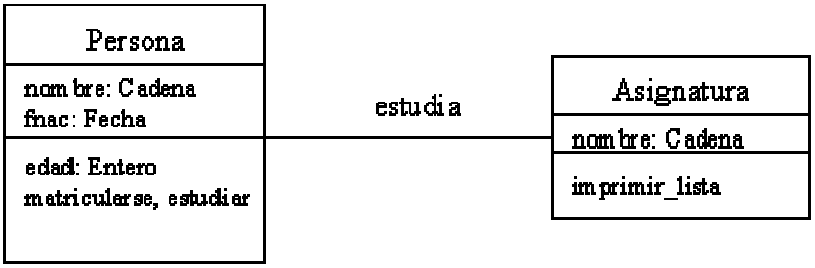
Las operaciones figuran en la tercer área del símbolo de las clases. Opcionalmente figuran también la lista de argumentos y el tipo de resultado de la operación (si es que la operación devuelve algún resultado). En los símbolos de instancia no figuran las operaciones.

Enlaces.

Un **enlace** es una conexión entre dos o más instancias (objetos). Los enlaces pueden ser considerados como las instancias de las asociaciones.

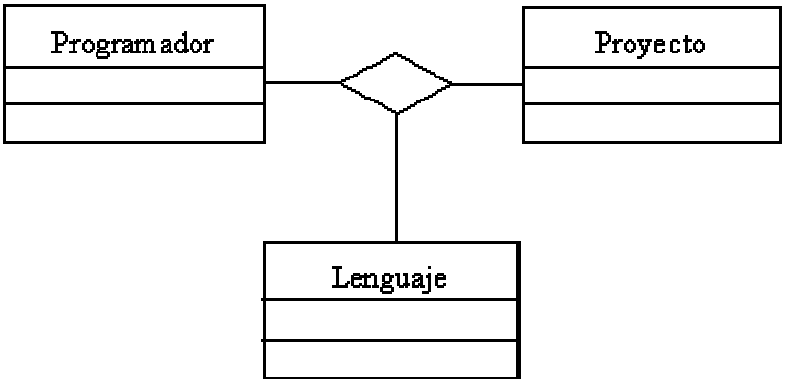
Asociaciones.

Una **asociación** es una abstracción de un grupo de enlaces con una estructura común (todos ellos conectan instancias de objetos de las mismas clases) y una semántica común (todos los enlaces tienen el mismo significado. Una asociación describe un conjunto de enlaces potenciales de la misma forma que una clase describe un conjunto de instancias potenciales.



Tanto los enlaces como las asociaciones se representan mediante arcos, generalmente etiquetados con el nombre de la asociación. Aunque la etiqueta induce a leer una asociación en un determinado sentido, las asociaciones pueden recorrerse en cualquier dirección.

Los enlaces y las asociaciones pueden conectar más de dos objetos. En este caso se representan mediante un rombo con conexiones a cada uno de los objetos:



La mayoría de las asociaciones conectan solamente dos clases de objetos y, siempre que sea posible usaremos asociaciones binarias porque son más sencillas de nombrar, entender y de implementar Sin embargo habrá casos en los que no podamos desglosar una asociación n-aria en varias asociaciones binarias. En el ejemplo anterior si desglosamos la asociación en tres (*Programador - conoce - Lenguaje, Proyecto - se implementa en - Lenguaje, y Programador - participa en - Proyecto*), no podemos representar que un *Programador* participa en un *Proyecto* programando en un *Lenguaje* determinado de lo que conoce.

Multiplicidad.

Cada asociación puede modelar la conexión un número indeterminado de objetos de las clases que conecta. Para representar el número de instancias de cada clase que pueden participar en una asociación utilizaremos la siguiente notación en cada extremo de la asociación:

¡ Opcional. La asociación puede relacionar 0 ó 1 instancias de la clase

· Muchos. Significa de 0 a N.

3 Exactamente 3.

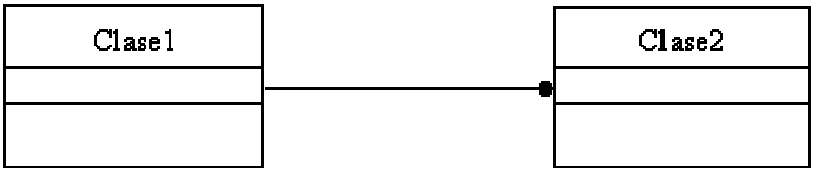
2,4 Dos o cuatro.

2-4 De dos a cuatro.

4+ Más de cuatro.

Exactamente 1.

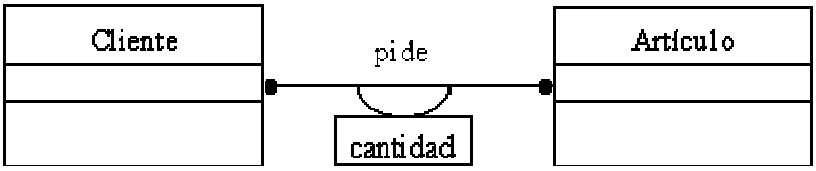
Una multiplicidad mayor que uno (p. ej. la existente entre las clases *Clase1* y *Clase2* abajo) indica que para cada instancia del *Clase1* pueden existir muchas instancias de la asociación (muchos enlaces) que lo conecten a instancias de la *Clase2*.



Con frecuencia la multiplicidad no viene expresada en la especificación preliminar, por lo que tendremos que decidirla en base al uso que queremos hacer del sistema. No es conveniente abusar de relaciones del tipo muchos a muchos pues, aunque son las más generales, van a sobrecargar el sistema a la hora de la implementación. De todas formas, no hay que preocuparse mucho de la multiplicidad en las etapas iniciales del análisis, cuando tengamos un mayor conocimiento del dominio del problema será más fácil decidir sobre ella.

Atributos de una asociación.

Al igual que los objetos contienen atributos, las asociaciones pueden contenerlos también. (p. ej. asociación *pide* entre *Clientes* y *Artículos*, que tiene un atributo *cantidad*). Estos atributos no pertenecen a ninguna de las clases relacionadas por lo que no hay más remedio que representarlos en la asociación. Estos atributos tendrán un valor propio para cada una de las instancias de la asociación (enlaces).



No obstante, si la asociación tiene multiplicidad 1/1 o 1/M podemos incluir estos atributos en la clase que tiene multiplicidad 1. Por ejemplo, en la relación *Persona - compra - Casa*, el atributo *fecha de compra*. puede incluirse en la clase *Casa*. Sin embargo, si la asociación es M/M, no podemos hacer esto: por ejemplo en un sistema de multipropiedad una casa puede pertenecer a varias personas, que la compran en fechas distintas. En cualquier caso no es conveniente incluir los atributos de las asociaciones en alguna de las clases relacionadas, puesto que reducimos la flexibilidad de nuestro modelo, que tendrá que ser reestructurado en caso de que necesitemos cambiar la multiplicidad.

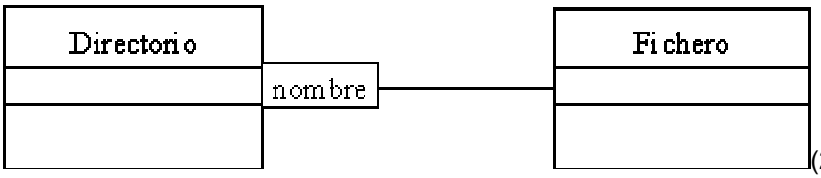
Una asociación con atributos se parece bastante a una clase. En determinados casos (si la asociación contiene también operaciones o si mantiene asociaciones con otras clases podemos modelarla como una clase.

Calificación.

Un tipo especial de asociaciones son las asociaciones calificadas. Si queremos expresar mediante el modelo que *'un directorio contiene varios ficheros, cada uno de ellos identificados mediante un nombre'* podemos hacer una asociación 1/M normal, con *nombre fichero* como atributo de *Fichero* (1)

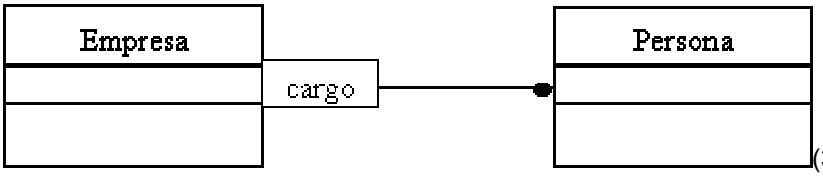
(1)

De esta forma, partiendo del directorio podemos acceder a todos sus ficheros mediante la asociación múltiple, pero así no expresamos que el nombre **identifica** a los ficheros dentro del directorio. La solución es utilizar *nombre fichero* como calificador en el extremo opuesto al M de la asociación (2), en este caso desaparece la multiplicidad en la asociación, puesto que utilizando el nombre la asociación múltiple se convierte en unitaria y, además, el modelo contiene una información mucho más exacta.

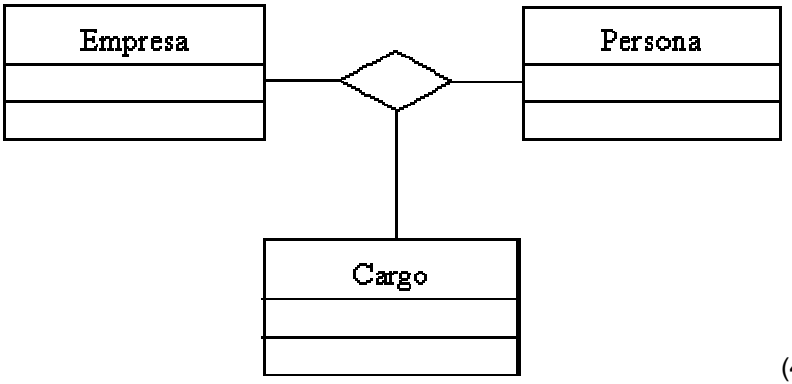


(2)

La calificación reduce la multiplicidad pero en algunos casos no se logra conseguir una multiplicidad unitaria. Tomemos como ejemplo el calificador *cargo* en la plantilla de una empresa. Suponemos que varias personas pueden ocupar el mismo cargo (3)



(3)

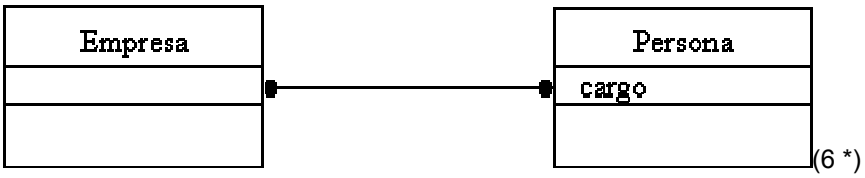


(4)

En algunos casos podemos considerar al calificador como un atributo especial del objeto situado al otro extremo de la asociación (1), o también podemos considerar la asociación calificada como una forma de asociación ternaria (4) o asociación con atributos (5). Por último si consideramos que una misma persona puede ocupar varios cargos distintos en la empresa, no podemos considerar *cargo* como un atributo de persona (6 *), pues esto significaría que una persona ocupa varias veces el mismo cargo en varias empresas.

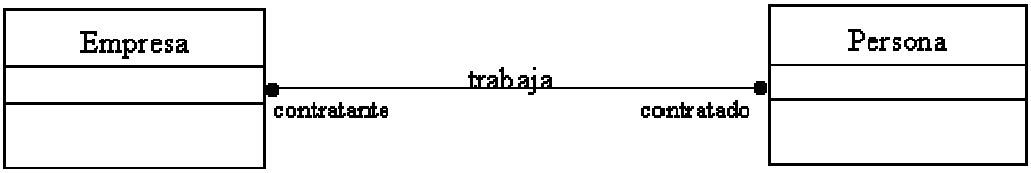
(5)

En resumen, la calificación nos permite reducir la multiplicidad y tiene una mayor potencia expresiva que las otras dos opciones (considerar el calificador como atributo de una de las clases o incluirlo como atributo de la asociación) por lo que optaremos por la asociación calificada cuando nos encontremos ante situaciones similares a las descritas.



Roles o papeles.

Para mayor claridad podemos etiquetar los extremos de las asociaciones con el rol o papel que representa la clase en la asociación. Así, en la relación *trabaja,* las personas juegan el papel de empleados mientras que las empresas juegan el papel de contratantes.

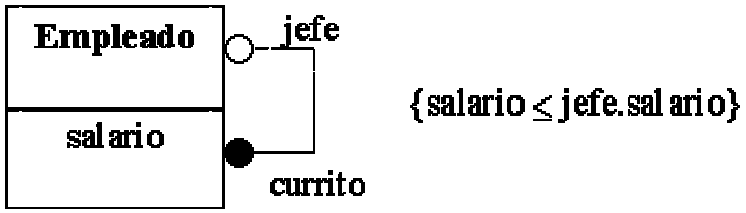


Restricciones

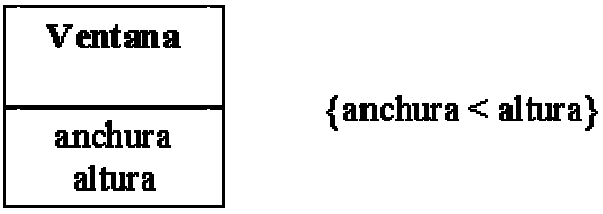
Las restricciones son relaciones funcionales entre entidades de un modelo de objetos. El término entidad incluye objetos, clases, atributos, enlaces y asociaciones. Una restricción restringe los valores que una entidad puede tomar.

Ejemplos:

- Restricción entre objetos: El salario de un empleado no puede superar el salario del jefe.



- Restricciones entre atributos de un objeto: Una ventana tiene que ser más alta que ancha.

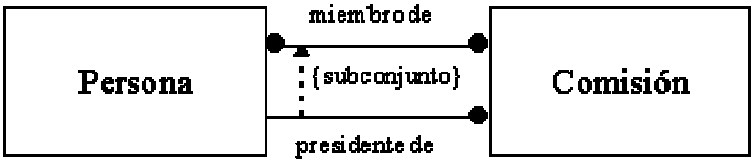


- Restricciones sobre un objeto a lo largo del tiempo: la prioridad de un trabajo no puede aumentar.

- Restricciones sobre enlaces:
    - La multiplicidad restringe el números de objetos relacionados con un objeto determinado.
    - Por regla general, las instancias conectadas al extremo M de una asociación no tienen ningún orden, pero en algunos casos sí que existe este orden. Un ejemplo de relación ordenada es la que se establece entre un documento y sus páginas. Para expresar esta característica etiquetaremos el extremo M con **{ordenado}** .

Las restricciones simples pueden situarse en el modelo de objetos; restricciones complejas aparecerán en el modelo funcional. Las restricciones no tienen porqué aparecer inicialmente en el modelo de objetos, estas irán añadiéndose conforme se vaya concretando en la definición del modelo.

Las restricciones se escriben entre llaves y se colocan junto a la entidad restringida. Las restricciones se expresan en lenguaje natural o con ecuaciones. En caso de que existan varias entidades involucradas en una restricción, dibujaremos una línea de puntos entre las entidades. (Ejemplo: una asociación puede ser subconjunto de otra: el presidente de una comisión debe ser un miembro de la comisión -la asociación *presidente_de* es un subconjunto de la asociación *miembro_de*).



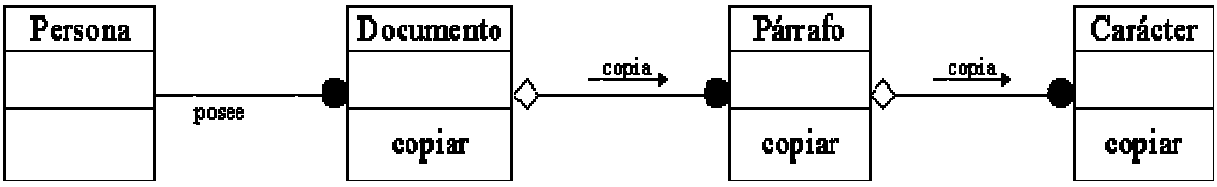
Relaciones de composición o agregación.

La composición nos permite representar las relaciones de *se compone de* o *es una parte de* que existen entre los objetos del modelo. Podemos utilizar asociaciones normales para representar la composición (y así lo hemos hecho en alguno de los ejemplos anteriores) pero la relación de composición tiene unas características especiales, por lo que se ha incluido un tipo de asociación especial para representarla.

Estas características son que la composición es transitiva, mientras que las relaciones normales no lo son. Por otra parte, la composición es antisimétrica y, por último, algunas propiedades del conjunto se propagan o influyen a las partes (p. ej. Un robot móvil con un brazo articulado: la posición del manipulador en el extremo del brazo depende no sólo de la posición (ángulos de rotación del brazo) sino también de la posición de todo el robot móvil.

La relación de composición se expresa gráficamente con un rombo al extremo de la asociación (p. ej. los *Documentos* se componen de *Párrafos*, que a su vez se componen de *Caracteres*). Para modelar un objeto que se compone de objetos de varias clases distintas (p. ej. una cebra que se compone de una cabeza, un cuerpo, hasta cuatro patas y

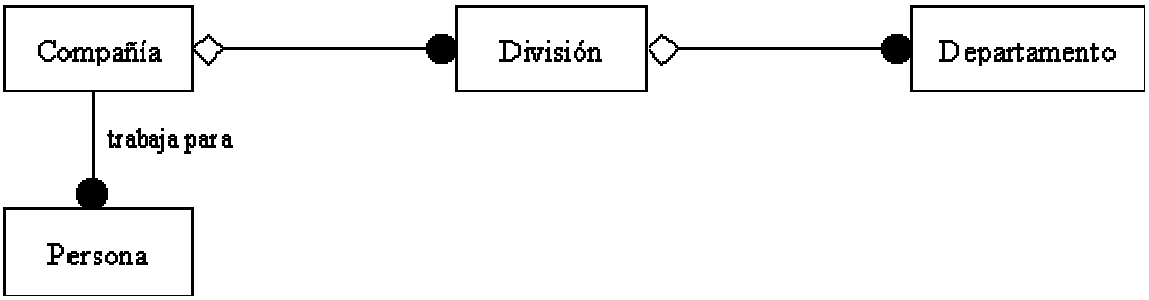
una cola opcional) podemos agrupar las relaciones de composición en una única asociación ramificada, indicando la multiplicidad en cada extremo.



A veces, la aplicación de una operación a un objeto supone la aplicación automática de esta operación a un conjunto de objetos. A esto se le llama propagación.. La operación de *copiar* se propaga del documento a sus párrafos y de estos a sus caracteres. La operación sin embargo no se propaga en la dirección inversa, el hecho de *copiar* un párrafo no implica la copia del documento.

Agregación y Asociación

Según lo visto, la agregación no es más que un tipo (frecuente) de asociación, que tiene unas características determinadas y que se representa en OMT mediante una notación gráfica especial. La diferencia entre asociación y agregación es fundamentalmente semántica, y hay que tener esto presente a la hora de decidir modelar una relación entre clases como asociación o como agregación. Por ejemplo, una *Empresa* es una agregación de *Divisiones*, que a su vez son agregaciones de *Departamentos*. Sin embargo, no es una agregación de *Personas*, puesto que *Empresa* y *Persona* son objetos independientes.

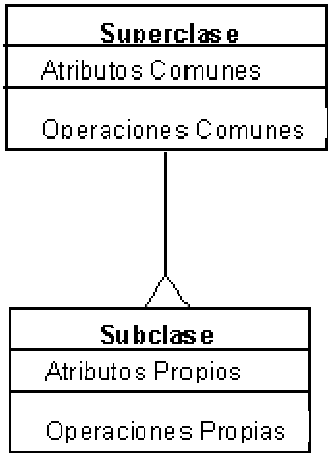


Generalización.

El último tipo de relación puede presentarse entre las clases es la relación de generalización o especialización (depende desde dónde se mire).

La generalización es una forma de abstracción que permite modelar que varias clases comparten una serie de características comunes, a la vez que tienen características propias. En el paradigma OO la generalización se relaciona con el mecanismo de **herencia**, mediante el que los objetos heredan las características comunes (atributos y operaciones) y es la fuente principal de la reutilización de código.

Generalización o **especialización** es la relación que se establece entre una clase y una o más versiones refinadas de esta clase. La clase general se denomina superclase y las clases especializadas, subclases.



En la superclase definiremos los atributos y operaciones comunes a todas las subclases. En cada subclase definiremos los atributos y métodos específicos para esa subclase. El mecanismo de herencia garantiza que todas las subclases heredan los atributos y operaciones definidos en la superclase.

La relación de generalización puede modelarse en varios niveles y es transitiva: las instancias de una subclase heredan los atributos y operaciones de todas las clases antecesoras. Además, cada instancia de una subclase puede considerarse también una instancia de cada una de las clases antecesoras: una lista ordenada doblemente encadenada es también una lista doblemente encadenada y una lista.

Si hay varias clases que particularizan una dada, podemos modelar esto mediante una sola relación de generalización ramificada. En ocasiones, existe un atributo de la superclase cuyos valores determinan la adscripción de un objeto a una u otra de las subclases. Un atributo de este tipo se denomina **discriminante.**

Redefinición.

Las subclases no sólo heredan los atributos y las operaciones de la superclase sino que también pueden redefinirlos. Esto se hace cuando un método general definido en una superclase no puede aplicarse tal como está a algunas de las subclases o puede realizarse una implementación más eficiente del método (p. ej. en la jerarquía anterior podemos redefinir el método *girar*. De la misma forma, las clases que hereden de *Polígono* redefinirán el método *área*.

La redefinición no puede cambiar la semántica de la característica redefinida (la operación tiene que ser la misma, aunque el método cambie) ni puede cambiar la **signatura** (es decir la forma: el número y tipo de los parámetros) de la característica redefinida. Algunos lenguajes de POO permiten que la redefinición restrinja el tipo de los parámetros (que sea por ejemplo un subrango).

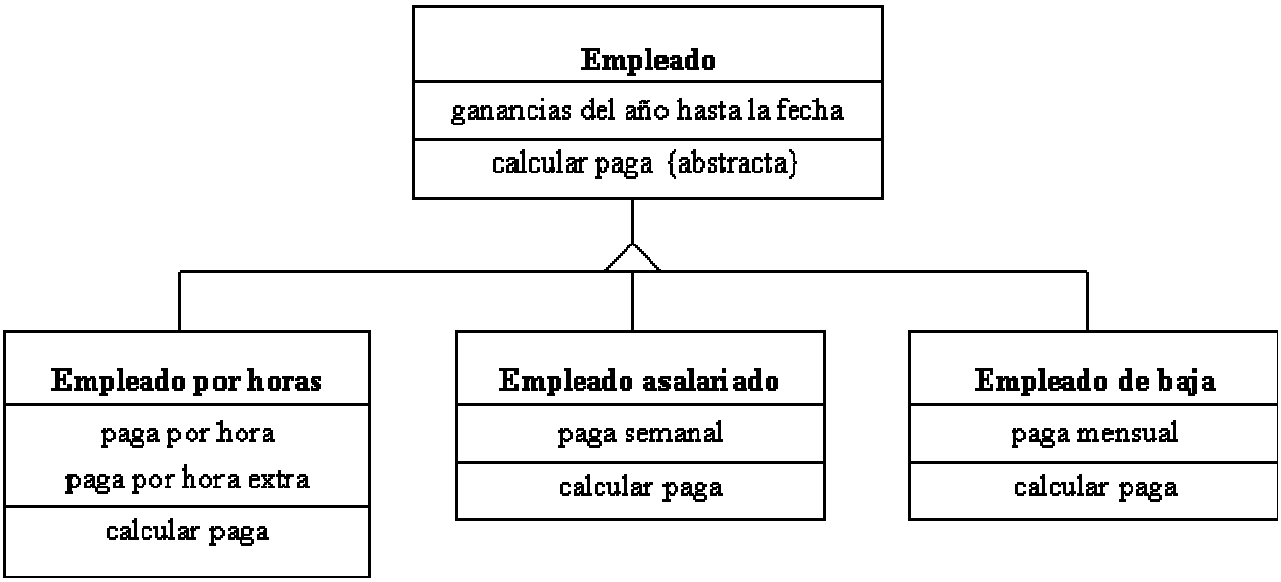
Clases Abstractas

Una clase abstracta es una clase que no tiene instancias directas pero cuyas clases descendientes (clases concretas) sí pueden tener instancias directas. Una clase concreta puede tener subclases abstractas, pero éstas han de tener subclases concretas (las hojas de una jerarquía de clases han de ser clases concretas).

Las clases abstractas organizan características comunes a varias clases. A veces es útil crear una superclase abstracta que encapsule aquellas características (atributos, operaciones y asociaciones) comunes a un conjunto de clases. Puede ser que esta clase abstracta aparezca en el dominio del problema o bien que se introduzca artificialmente para facilitar la reutilización de código.

Normalmente las clases abstractas se usan para definir métodos que son heredados por sus subclases. Sin embargo, una clase abstracta puede definir el protocolo de una determinada operación sin proporcionar el correspondiente método. Esto de denomina operación abstracta, que define la forma de una operación para la que cada subclase concreta debe suministrar su propia implementación. Una clase concreta no puede tener operaciones abstractas

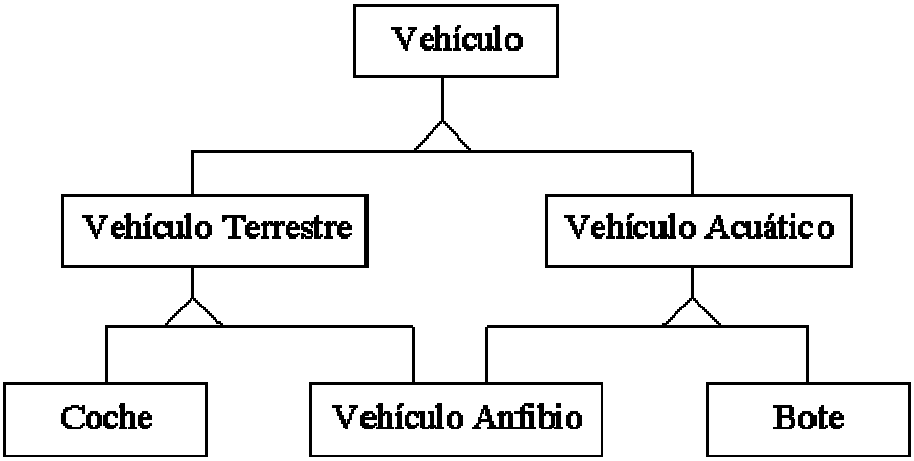
porque los objetos de esta clase tendrían operaciones indefinidas. Para indicar que una operación es abstracta se coloca una restricción {abstracta} junto a ella.



Herencia múltiple

La herencia múltiple permite a una clase tener más de una superclase, y así heredar las características de todos sus padres. Esto complica las jerarquías de herencia, que dejan de ser árboles para convertirse en grafos. La gran ventaja de la herencia múltiple es el incremento en las posibilidades de reutilización. El inconveniente es la pérdida de simplicidad conceptual y de implementación.

La herencia múltiple presenta dos problemas: conflictos y herencia repetida. Cuando una clase hereda simultáneamente sus propiedades de varias clases, pueden existir en estas propiedades representadas por el mismo identificador con significados diferentes; a esto se le denomina *conflicto*. Los conflictos que provoca la herencia múltiple se agravan en presencia de situaciones en las que una clase es heredada por otra por medio de caminos distintos; es lo que se denomina *herencia repetida*.



Existen muchas propuestas para resolver este tipo de problemas, casi tantas como lenguajes orientados a objetos. Podemos encontrar desde sistemas que dejan al usuario su resolución (algunas versiones de Smalltalk), hasta los que prohiben situaciones conflictivas (C++, Eiffel), pasando por sistemas que usan algún heurístico linealizando el árbol de herencia (CommonLoops), o los que simplemente evitan el problema impidiendo la herencia múltiple (Java).

Las ventajas e inconvenientes de las posibles alternativas de la herencia múltiple se han estudiado con cierta profundidad, llegando a la conclusión de que cada alternativa tiene razones a favor y en contra, lo que impide tomar decisiones de diseño que satisfagan todo tipo de exigencias. En la mayoría de los casos, la interpretación que se da a la herencia múltiple depende más de su adecuación a la implementación que a otros motivos.

Consejos prácticos.

Por último, una serie de consejos prácticos para realizar modelos de objetos. Muchos de estos consejos son muy similares a los de los métodos estructurados: lo importante de una notación gráfica es que represente de forma correcta y clara el dominio de aplicación.

- **No lanzarse a dibujar clases y asociaciones sin sentido.** Primero hay que entender el problema. Los métodos y herramientas no hacen análisis, el análisis lo hacemos nosotros ayudándonos con estos métodos y herramientas.

- **Intentar que el modelo sea simple, evitando complicaciones innecesarias.** De lo que se trata es de que el modelo sea claro, que alguien más pueda entenderlo, aceptarlo o criticarlo.

- **Los nombres de objetos, asociaciones, atributos y operaciones deben ser significativos**. Antes de poner un nombre hay que pensárselo un poco: el nombre representa al elemento. Las clases deben nombrarse con sustantivos y las asociaciones con verbos. Por las asociaciones se envían mensajes, pero no objetos. Un nombre largo puede ser más significativo pero no será más claro.

- **No incluir en los objetos punteros a otros objetos**. Esto sólo sirve para que las relaciones entre objetos queden ocultas. Hay que modelar estas relaciones mediante asociaciones. Los punteros es como implementan algunos lenguajes las asociaciones, pero nuestro modelo debe ser independiente de la implementación.

- **Utilizar, si es posible, asociaciones binarias**. Las asociaciones de orden superior son más difíciles de nombrar, entender e implementar. De todas formas hay asociaciones que no podemos descomponer en binarias sin perder información.

- **Dejar la definición de la multiplicidad para cuando se tenga un mejor conocimiento del problema**. Pero no hay que olvidarse de ella. No hay que abusar de la multiplicidad 1/M o M/M pues, aunque sea más general la implementación es menos eficiente. Por otra parte, hay que tener cuidado con la multiplicidad 1/1. En muchos casos se trata de multiplicidad opcional (es decir, 1/0-1). La multiplicidad 1/1 implica que los objetos conectados no pueden existir de forma independiente, sino sólo por pares. Normalmente esta consideración es excesivamente estricta y puede llevarnos a perder información).

- **No incluir los atributos de las asociaciones en las clases**. Aunque las asociaciones sin atributos son más sencillas, el hacer esto limita la flexibilidad del modelo.

- **Utilizar preferentemente asociaciones calificadas en vez de ternarias o con atributos**. Proporcionan una información más exacta del dominio de aplicación.

- **Evitar las jerarquías de composición o generalización de muchos niveles**. Dificultan la legibilidad del modelo.

- **Revisar el modelo hasta que sea satisfactorio**. No podemos pretender que la primera versión sea correcta. Es conveniente mostrar el modelo a otras personas (no en el examen).

- **Documentar el modelo**. Una imagen vale más que mil palabras, pero una imagen con texto es siempre mucho más explicativa que otra sin él.

- **Utilizar sólo los elementos necesarios**. No se trata de lucirse y mostrar que se dominan todos los conceptos del modelo, sino de que el modelo sea correcto y claro.

Modelo dinámico.

Diagramas de estados.

El modelo dinámico del método OMT se corresponde con el modelo de control o modelo de comportamiento de las técnicas de análisis estructurado. En ambos casos, el modelo se representa mediante DEs, pero en el caso de OMT estos DEs se utilizan para modelar el comportamiento de cada clase de objetos, es decir, para modelar el comportamiento común a todas las instancias de una clase.

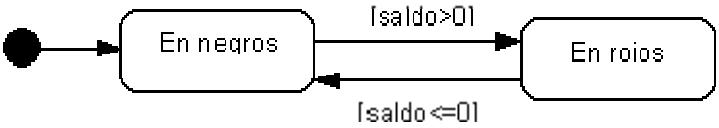
Al igual que todas las instancias de una clase comparten atributos, pero tienen valores particulares para esos atributos, todas las instancias de una clase se comportan de igual forma, pero pueden encontrarse en estados distintos.

Los elementos que figuran en un DE ya son conocidos, aquí sólo vamos a referirnos a las diferencias con respecto al análisis estructurado.

Estados y eventos.

Por *estado* de un objeto entendemos los valores de los atributos y los enlaces que mantiene un objeto en un momento determinado. Los objetos interactúan unos con otros y como consecuencia de esas interacciones cambian de estado (es decir, cambian el valor de sus atributos o sus enlaces con otros objetos).

Los estados que figuran en los DEs son abstracciones de los valores de los atributos y enlaces de un objeto, es decir, engloban conjuntos de valores de los atributos, de forma que muestran situaciones en las que el objeto presenta un determinado comportamiento. Por ejemplo, en el objeto *Cuenta Corriente*, el estado *En rojos* engloba todos los estados del objeto en los que el saldo es negativo.



Al definir los estados, debemos ignorar los atributos que no afectan al comportamiento del objeto y agrupar todas las combinaciones de valores de atributos y enlaces en las que el objeto presente el mismo comportamiento.

La interacción entre objetos se realiza mediante *eventos*. Como consecuencia de (la recepción de) un evento, un objeto puede realizar una serie de acciones y/o enviar eventos a otros objetos y/o cambiar de estado. Los eventos son el método de intercambiar información entre los objetos. Esta información consistirá en la mayoría de los casos en una señal lógica, pero en otros puede tratarse de información más compleja, y el evento tendrá una serie de atributos:

cancelar evento lógico

retirar(numero de cuenta, 5000) evento con atributos

La respuesta de un objeto en un estado determinado ante un evento determinado puede depender de los valores exactos de los atributos, pero sólo cuantitativamente. Cualitativamente, la respuesta será la misma para todos los objetos que se encuentren en ese estado.

La relación o secuencia de estados, eventos, transiciones y acciones se representa en los DEs. El modelo dinámico es un conjunto de DEs, uno para cada clase de objetos que tenga un comportamiento dinámico no trivial (no es necesario hacer un DE para cada clase sino sólo para aquéllas que lo precisen). La relación entre los distintos DEs se establece mediante los eventos que emiten unos objetos y consumen otros (se emiten en un determinado DE y se consumen en otro).

Guardas.

La encapsulación propia del paradigma OO impide con carácter general el acceso directo a valores de los atributos de otros objetos. Por este motivo una guarda es una condición que se define sobre los **atributos propios** de un objeto . Al recibir un evento, la transición etiquetada con dicho evento se dispara si y sólo si se satisface la guarda.

Actividades y acciones.

Las actividades y las acciones son las respuestas que tiene que dar un objeto ante los eventos que se producen en el exterior o las guardas (condiciones de datos) que se satisfacen internamente. Típicamente, las acciones y las actividades van a ser operaciones definidas en la clase a la que pertenece el objeto, aunque en ocasiones las acciones pueden consistir en enviar un mensaje a otro objeto.

Como ya sabemos, una **actividad** es una operación que necesita un tiempo para completarse, y se corresponde siempre con la ejecución de un determinado método o con el objeto inactivo, esperando a que se cumplan determinadas condiciones. Cuando un objeto entra en un estado, ejecuta la actividad correspondiente hasta que ésta finalice o bien hasta que el disparo de una transición nos haga abandonar dicho estado.

Por el contrario, una **acción** es una operación instantánea, que idealmente no necesita tiempo para realizarse y que se asocia, por tanto, al disparo de una determinada transición. Las acciones generalmente consisten en asignar valores a los atributos del objeto o en generar eventos para que sean tratados por otro proceso (con la notación *enviar: evento*). Por este motivo, las acciones no suelen corresponderse con operaciones del objeto, sino posiblemente con parte del código de ciertas operaciones.

Trazas de eventos.

Los eventos son el medio que utilizan los objetos para comunicarse entre sí. Podemos representar la secuencia de eventos que se ha producido entre los objetos de un determinado sistema mediante **una traza de eventos**. Esta traza es un diagrama que muestra la secuencia de envíos y recepciones de eventos entre varios objetos.

Las trazas de eventos son útiles para mostrar cómo un sistema ha llegado a un escenario determinado, y se mostrarían entonces junto con el diagrama de instancias que represente ese escenario, pero también son útiles para construir el modelo dinámico del sistema, pues muestran una secuencia lógica de eventos que podemos utilizar para realizar los DE de cada clase. A partir de varias trazas de eventos podemos construir el DE de una clase.

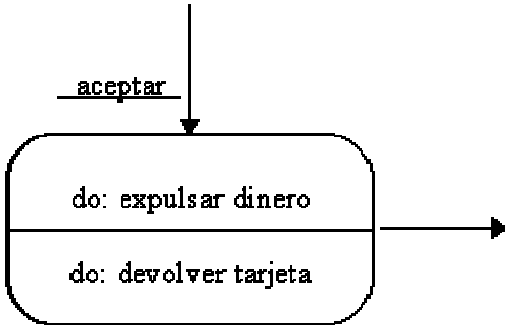
Cambiando el punto de vista, un diagrama de estados nos permite determinar la secuencia de estados por los que pasa un objeto a partir de una traza de eventos. Si el evento recibido figura en alguna de las transiciones que salen del estado, entonces la transición **se dispara** y el objeto pasa al estado de llegada de la transición.

Composición en los DEs.

Una de las características del paradigma OO es la definición de objetos componiendo otros objetos. El DE de un objeto compuesto estará formado por la unión de los DEs de cada uno de los objetos que lo componen. Cada uno de los objetos estará en un estado determinado, por lo que el estado del objeto compuesto estará representado mediante una tupla que contiene cada uno de los estados de los componentes. Es decir, los objetos compuestos tienen estados compuestos.

Esta posibilidad nos permite describir la concurrencia, no sólo en los objetos compuestos, sino en cualquier objeto. Podemos descomponer un objeto simple en varias partes, cada una que contiene un subconjunto de sus atributos y enlaces, y realizar un DE para cada uno de esos subconjuntos. Al igual que antes, el estado del objeto vendrá representado por la composición de estos DEs. Cada uno de los componentes de un objeto modelado mediante un DE compuesto evoluciona de forma concurrente al resto.

En algunos casos, podemos necesitar mostrar la concurrencia entre las actividades que se realizan dentro de un estado. Podemos encontrarnos con un estado en el que la actividad a realizar pueda ser descompuesta en una serie de pasos concurrentes. En este caso dividiremos el estado en dos (o más) subestados concurrentes.



Ejemplo: Las operaciones *Expulsar Dinero* y *Devolver Tarjeta* del cajero automático.

La sincronización de las transiciones de salida de cada uno de los subestados se hace juntando estas transiciones para formar la transición de salida del objeto compuesto.

Generalización en los DEs.

Generalización de estados.

Puede ocurrir que el objeto tenga un comportamiento tan complejo que el DE que lo modele sea demasiado grande y difícil de leer. En estos casos, podemos hacer DEs anidados, en los que un estado del DE padre se corresponda con un DE hijo. En este DE hijo se detalla el estado padre en una secuencia de estados, transiciones, acciones y actividades. Esta forma de anidamiento es similar al anidamiento que se producía en los DEs del análisis estructurado.

Podemos utilizar esta posibilidad en el caso de que la actividad que se asocie a un estado pueda ser descompuesta en una serie de operaciones secuenciales. En el DE hijo representaremos cada una de estas operaciones en un nuevo estado. Las transiciones que entran y salen del estado padre deben figurar en los estados iniciales y finales del DE hijo.

Alternativamente, podemos utilizar el anidamiento de estados para representar las transiciones comunes a una serie de estados, al igual que hacíamos en el análisis estructurado. Cada estado anidado o **subestado** comparte las transiciones de salida del estado padre o **superestado**, por otra parte las transiciones que entran en el padre nos llevan al estado inicial del DE anidado.

Cuando un objeto esté en un superestado, habrá de estar necesariamente en uno y sólo en uno de los subestados correspondientes.

Generalización de eventos.

Al igual que los estados también podemos generalizar los eventos. Podemos agrupar los eventos en clases hasta conseguir una jerarquía de eventos, lo que nos permite utilizar diferentes grados de abstracción en diferentes partes del modelo.

Herencia de DEs.

Según hemos dicho a cada clase del modelo de objetos se asocia un DE que modela su comportamiento, pero ¿qué sucede cuando dos clases están relacionadas mediante generalización? Entre las propiedades que se heredan de la clase padre está también el comportamiento, por lo que las subclases **heredan el DE de la superclase.**

Es posible que la subclase añada atributos u operaciones con respecto a la superclase. Entonces podemos considerar el DE de la subclase como la composición del DE de la superclase con el/los DEs que modelen el comportamiento del objeto con respecto a los nuevos atributos, al igual que hacíamos con los objetos compuestos.

En otros casos, la especialización del DE consistirá en el refinamiento de alguno de los estados del DE de la superclase, descomponiendo este estado en el DE de la subclase. Esto se corresponde con una división más detallada del espacio de estados heredado (el conjunto de valores de los atributos heredados).

En cualquier caso existe la posibilidad de conflicto: si la subclase redefine el comportamiento de los objetos respecto a alguno de los atributos heredados. No se pueden incluir transiciones o estados nuevos en el DE heredado puesto que entonces no habría una correspondencia entre ambos DEs. Estas situaciones, en las que las clases herederas modifican tienen un comportamiento distinto al de la clase padre y no es posible establecer una correspondencia entre ambos, son bastante frecuentes en la práctica. En estos casos no podemos hablar de herencia del DE sino de una redefinición completa del mismo. Esto es una limitación del modelo que no está resuelta y que contrasta con las posibilidades de redefinición de operaciones y atributos al especializar una clase. Esta limitación puede presentarse también cuando el comportamiento relacionado con los atributos nuevos definidos en la clase hija no puede modelarse de forma independiente a los heredados, sino que requiera realizarse de forma conjunta.

Modelo dinámico: consejos prácticos.

- **No todas las clases necesitan un DE**. Sólo haremos DEs para aquellas clases que lo necesiten. Habrá clases cuyo comportamiento sea trivial, por ejemplo aquéllas clases que cuyo comportamiento sea siempre igual, es decir, que posean un único estado.

- **Utilizar trazas de eventos como ayuda para construir el DE**. Las trazas de eventos describen una secuencia lógica de eventos en el sistema. Podemos utilizarlas para construir el esqueleto del DE, empezando por la secuencia de operaciones más habitual en el sistema. Luego debemos ir ampliando para que contemple otras secuencias cada vez menos frecuentes, no olvidándonos de las secuencias de error.

- **Para definir estados, sólo hay que tener en cuenta los atributos que influyen en el comportamiento**. Los estados se definen a partir de subconjuntos de valores de los atributos. Sólo consideraremos los atributos que influyen en el comportamiento (los nombres no) y agruparemos los valores de estos atributos en conjuntos de forma que los objetos que estén en un estado presenten un comportamiento común.

- **Decidir la granularidad de estados y eventos según las necesidades del modelo**. Los estados se asocian a actividades, que normalmente pueden ser descompuestas en fases, con varios niveles de granularidad. Según esto podemos hacer también los DEs con distinto tamaño de grano. Escogeremos el más adecuado según las necesidades de la aplicación y el grado de detalle que le queramos dar al modelo.

- **Distinguir entre eventos y guardas**. Los eventos son mensajes que utilizan los objetos para comunicarse. En los DEs los eventos que recibe un objeto aparecen en las transiciones e indican que transición se dispara cuando se recibe un evento. Las guardas son condiciones que se establecen sobre valores de los atributos del propio objeto del que estamos haciendo el DE.

- **Distinguir entre actividades y acciones**. Las actividades son operaciones que tienen una duración, por lo que se representan en los estados. Las acciones son operaciones que (idealmente) se realizan instantáneamente, y se representan en las transiciones.

- **Utilizar diagramas anidados cuando una transición se aplique a varios estados**. Esto nos permite *reutilizar* las transiciones aplicando los principios de generalización y especialización y la regla de herencia a los DEs.

  En los diagramas anidados hay que tener en cuenta la consistencia, al menos de las transiciones de salida. Las transiciones de salida del estado padre deben figurar en alguno de los estados del DE hijo. Las de entrada no hace falta pues suponemos que el DE hijo comienza siempre por el estado inicial.

- **Realizar un DE compuesto si:**

¡ **Existen relaciones de composición entre los objetos**. El DE del objeto compuesto estará formado por el conjunto de los DEs de los componentes. El estado de un objeto compuesto es una tupla, cuyos elementos son los estados de los objetos componentes.

¡ **Existe concurrencia interna en el objeto**. Si el objeto es concurrente (si puede o debe realizar varias acciones al mismo tiempo) podemos dividir los estados en subestados concurrentes, en cada uno de ellos se realiza una actividad.

¡ **Podemos dividir el comportamiento en facetas independientes**. Si el comportamiento presenta facetas que dependen de conjuntos disjuntos de atributos, podemos modelar este comportamiento mediante varios DEs, uno para cada conjunto. Estos DEs serán mucho más sencillos puesto que sólo van a manejar un subconjunto de los eventos, sin que tengamos que preocuparnos de definir transiciones para el resto.

¡ **Existen relaciones de herencia entre los objetos**. Si el objeto hijo define nuevos atributos, haremos un DE para definir el comportamiento según los valores de estos atributos nuevos. Si el objeto hijo descompone uno o más estados del padre, tendremos un DE anidado.

- **En caso de herencia de entre clases, el DE de la subclase no puede añadir estados o transiciones al DE padre**. Como un objeto puede ser visto tanto como una instancia de la clase padre como una instancia de la clase hijo, debe existir una correspondencia entre los estados y transiciones de los DEs de ambas. Si añadimos un estado en el DE del hijo ya no existe esa correspondencia, y un objeto en ese nuevo estado, considerado como una instancia de la clase padre, estaría un estado indeterminado.

- **Comprobar la consistencia de los eventos que se generan en un DE y se utilizan en otros**. Los eventos que se utilizan para disparar las transiciones de un DE deben ser generados por algún objeto. Los eventos que genera un objeto deben utilizarse en algún otro DE.

Modelo funcional.

El modelo funcional describe las computaciones que se realizan en un sistema, mostrando cómo se derivan los valores de salida a partir de los de entrada. El modelo funcional muestra cómo se realizan las operaciones del modelo de datos y las actividades acciones de los DEs.

La existencia de un modelo funcional basado en el uso de DFDs distingue OMT de otras metodologías de desarrollo orientado a objetos. Sin embargo, no podemos considerar esto como algo positivo de OMT puesto que no es fácil la adecuación del modelo de procesos del análisis estructurado al tipo de código existente en un sistema orientado a objetos. Por este motivo apenas se usa y, de hecho, escritos recientes de Rumbaugh descartan el uso de DFDs para representar el modelo funcional del sistema, optando, en su lugar, por incluir casos de uso, diagramas de interacción de objetos y especificaciones de operaciones similares a las PSPECs del análisis estructurado.

La especificación de una operación puede realizarse mediante precondiciones y postcondiciones. Una precondición indica suposiciones sobre el estado del sistema al comienzo de la operación. Una postcondición describe el estado del sistema una vez realizada la operación. Las precondiciones y postcondiciones pueden expresarse mediante algún lenguaje formal, aunque en la mayoría de los casos basta con utilizar lenguaje natural, siempre que el significado quede claro.

Clase: Array[T]

Operación: ordenar

Función: reorganiza los elementos del array, de forma que queden en orden creciente.

Entradas: ninguna.

Salidas: ninguna.

Objetos modificados: el propio objeto.

Precondiciones

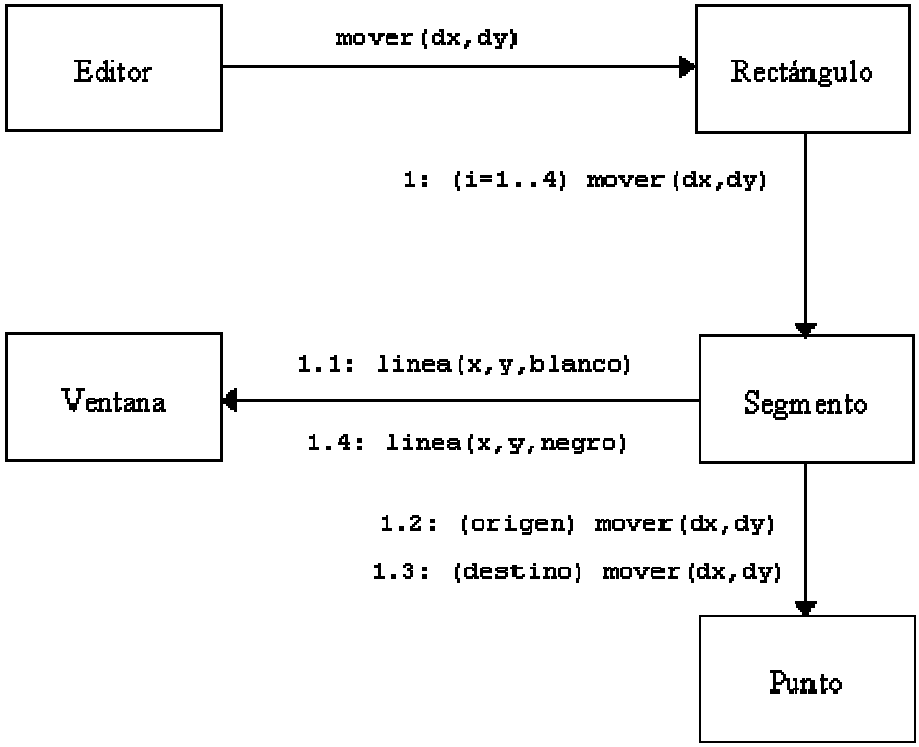
- Todos los elementos han de ser de tipo T o de un subtipo de T.
- El tipo T tiene una operacion *comparar*, que compara dos elementos y devuelve los valores LT, EQ, GT si el primero es menor, igual o mayor, respectivamente que el segundo. La operación *comparar* define una ordenación total de los valores de T.
- Se permite que el array tenga valores duplicados.

Postcondiciones

- Los elementos del array quedan ordenados de acuerdo a la función de comparación.
- El array contiene exactamente el mismo número de ocurrencias de cada valor de T que antes de ejecutar la operación.

En general, la implementación de las operaciones puede derivarse del modelo de comportamiento durante el diseño del sistema. Sin embargo, durante el análisis, será necesario describir, al menos, las operaciones de aquéllas clases para las que no se ha realizado diagrama de estados.

La implementación de una operación puede invocar otras operaciones, pero esto es un aspecto de diseño y no de especificación. Para mostrar la colaboración entre distintos objetos a la hora de realizar una operación determinada podemos realizar un *diagrama de interacción de objetos*.



5.7. Relación entre los modelos.

El modelo de objetos describe la estructura de datos sobre la que operan los modelos funcional y dinámico. Las operaciones en el modelo de objetos se corresponden con las acciones y actividades del modelo dinámico y con los procesos del modelo funcional.

El modelo dinámico describe el control de los objetos. El DE describe el comportamiento de los objetos de una clase, mostrando secuencias válidas de cambios de comportamiento en las clases del modelo de objetos. Los estados son clases de equivalencia de los valores de atributos y enlaces del objeto, que normalmente se asocian a la ejecución de una determinada operación en el objeto. Los eventos aparecen también como operaciones en el modelo de objetos.

Un subestado refina los valores de atributos y enlaces que puede tener un objeto. Una jerarquía de estados de un objeto es equivalente a una jerarquía de restricciones sobre el objeto. Como los LOO no suelen permitir restricciones de valores sobre el tipo de los atributos heredados o los parámetros de las operaciones (no se puede cambiar la signatura, ni siquiera restringiéndola), el modelo dinámico es el sitio adecuado para representar estas restricciones.

En la mayoría de los LOO, las instancias no pueden cambiar la clase a la que pertenecen a lo largo de su vida, pero sí que pueden cambiar de estado. Por tanto, para modelar objetos que van a estar sujetos a restricciones cambiantes, modelaremos estas restricciones como estados en lugar de como clases.

Las jerarquías de objetos y de eventos son totalmente independientes. Los eventos sirven para intercambiar información entre objetos de clases distintas. Las transiciones se suelen representar mediante operaciones, cuyo nombre es el evento que dispara la transición.

El modelo funcional describe funciones invocadas por operaciones en el modelo de objetos o acciones y actividades en el modelo dinámico. Las funciones operan sobre los valores de datos especificados en el modelo de objetos. El modelo de objetos muestra las entidades que realizan o padecen las funciones. El modelo dinámico muestra la secuencia en la que se realizan las funciones.

En resumen:

- **Relaciones con el modelo de objetos.**

  El modelo funcional muestra las operaciones que se realizan en cada clase y los argumentos de estas operaciones. El modelo dinámico muestra los estados de cada objeto y las operaciones que éstos realizan al recibir eventos y cambiar de estado.

- **Relaciones con el modelo dinámico.**

  El modelo funcional muestra las definiciones de las acciones y actividades del modelo dinámico. El modelo de objetos muestra los objetos que sufren o realizan las acciones y actividades del modelo dinámico.

- **Relaciones con el modelo funcional.**

  El modelo de objetos muestra las entidades que realizan o padecen las funciones del modelo funcional. El modelo dinámico muestra la secuencia en que se realizan las funciones del modelo funcional.

Método de análisis.

El objetivo del AOO es modelar el mundo real, de forma que pueda ser entendido. Es necesario abstraer las características importantes del problema en primer lugar, dejando los detalles para más adelante. Un buen modelo de análisis debe indicar lo que hay que hacer, sin restringir cómo hay que hacerlo, evitando tomar anticipadamente decisiones de implementación.

El modelo de análisis se compone del modelo de objetos, que representa la estructura estática de la información, el modelo dinámico, que indica la secuencia de eventos, y el modelo funcional, que muestra las transformaciones de datos. No todos ellos tienen la misma importancia, y la necesidad de desarrollar cada uno de ellos depende del dominio del problema. Habrá casos en que no sea necesario realizar el modelo dinámico o el funcional. El modelo de objetos siempre es necesario si vamos a hacer AOO.

El análisis no es una actividad secuencial, sino que los modelos se construyen de forma iterativa, añadiendo nuevas características o modificando el modelo según se refinan los requisitos y también a partir del conocimiento del dominio de aplicación obtenido al construir cada uno de los modelos.

Estrictamente, podemos dividir la etapa de análisis dentro de la metodología OMT en tres fases: modelo de objetos, modelo dinámico y modelo funcional. Sin embargo, si consideramos la fase inicial de conceptualización, debemos incluir también los casos de uso.

Casos de uso.

Los casos de uso describen las distintas formas de utilizar el sistema, visto desde su exterior, es decir, desde el punto de vista del usuario. Para realizar los diagramas de casos de uso se puede proceder como se describe a continuación:

- **Establecer los límites del sistema**. Hay que determinar qué objetos forman parte del sistema, qué objetos interactúan con él y cuáles no. Los casos de uso consideran el sistema como una caja negra, es decir, no entran a describir su estructura interna.

- **Determinar los actores que interactúan con el sistema**. Un actor es un rol o papel que juega un objeto externo en su relación con el sistema. Para determinar los actores debemos observar los objetos físicos que interactúan con el sistema, que en muchos casos juegan múltiples roles. Por ejemplo, una misma persona puede ser Usuario, Operador y Administrador de un cierto sistema. Cada rol es un actor diferente.

- Para cada actor, **determinar las diferentes formas en las que usa el** sistema. Por cada una de esas formas tendremos un caso de uso.

  No debería haber demasiados casos de uso en un sistema. Entre 5 y 10 casos bastan normalmente para mostrar los usos principales del sistema. Si no es así, es necesario utilizar un menor nivel de detalle. En cada caso de uso se debe escoger un nivel de detalle similar.

- **Identificar el evento inicial** que comienza el caso de uso.

- **Determinar la condición de terminación** que finaliza el caso de uso. A menudo un caso de uso puede ser descrito con diferentes niveles de detalle.

- **Describir la situación típica** en la que ocurre el caso de uso. Si hay variaciones o excepciones, es necesario describirlas también. Basta con utilizar lenguaje natural, un caso de uso no necesita ser demasiado formal.

Modelo de objetos.

Empezaremos a modelar el sistema realizando su modelo de objetos. Este modelo muestra la estructura estática de los datos del mundo real y las relaciones entre estos datos. El modelo de objetos precede normalmente al dinámico y al funcional porque normalmente está mejor definido en la especificación preliminar, es menos dependiente de detalles de la aplicación, es más estable respecto a la evolución de la solución y es más fácil de entender que el resto.

Los pasos a seguir son los siguientes:

Identificar objetos y clases.

Durante el proceso de desarrollo aparecen tres categorías de objetos: objetos del dominio, objetos de aplicación y objetos internos.

Los **objetos del dominio** son significativos desde el punto de vista del dominio del problema. Existen de forma independiente a la aplicación y tienen sentido para los expertos del dominio.

Los **objetos de aplicación** representan aspectos computacionales de la aplicación que son visibles para los usuarios. No existen en el espacio del problema, solo tienen sentido en el contexto de la aplicación que vamos a desarrollar para solucionarlo. Sin embargo, no dependen exclusivamente de decisiones de diseño, puesto que son visibles al usuario y no pueden ser cambiados sin alterar la especificación de la aplicación. No pueden obtenerse analizando el dominio de la aplicación, pero sí pueden ser reutilizados de aplicaciones anteriores, incluso aunque sean de diferente dominio. Los objetos de aplicación incluyen controladores, dispositivos e interfaces.

Los **objetos internos** son componentes de la aplicación que resultan invisibles para el usuario. Su existencia se deriva de decisiones de diseño para implementar el sistema. No deben aparecer durante el análisis. Una parte importante del diseño consiste en añadir objetos internos para hacer factible la implementación del sistema.

Por tanto, en el modelo de objetos figurarán tanto objetos del dominio como objetos de aplicación. Su construcción puede ser realizada en dos fases:

En una primera fase podemos construir un modelo que represente los objetos del dominio del problema y las relaciones que existen entre ellos. Este modelo equivale a lo que se llama *modelo esencial* en la terminología del análisis estructurado.

En una segunda fase podemos construir un *modelo de aplicación*, completando el modelo anterior con objetos de aplicación. Para esto jugarán un papel muy importante los casos de uso desarrollados durante la conceptualización del problema.

Los objetos del modelo pueden ser tanto entidades físicas (como personas, casas y máquinas) como conceptos (como órdenes de compra, reservas de asientos o trayectorias). En cualquier caso, todas las clases deben ser significativas en el dominio de aplicación. Hay que evitar definir clases que se refieren a necesidades de

implementación como listas, subrutinas o el reloj del sistema. No todas las clases figuran explícitamente en la especificación preliminar, algunas están implícitas en el dominio de aplicación o son de conocimiento general.

Podemos empezar haciendo una lista de clases candidatas a partir de la especificación preliminar. Normalmente las clases se corresponden con nombres en este documento. No hay que preocuparse aún de establecer relaciones de generalización/especialización entre las clases. Esto se hace para reutilizar código y estas relaciones aparecen más claramente cuando se definan atributos y operaciones.

Una vez que tenemos una lista de clases candidatas hay que proceder a revisarla, eliminando las incorrectas, siguiendo los siguientes criterios:

- **Clases redundantes**. Si dos clases representan la misma información nos quedaremos con la que tenga un nombre más significativa, eliminando la otra. (p. ej. Cliente y Usuario).

- **Clases irrelevantes**. Podemos haber incluido clases que sean importantes en el dominio de aplicación, pero que sean irrelevantes en el sistema que pretendemos implementar. Esto depende mucho del problema concreto.

- **Clases demasiado generales**. Las clases deben ser específicas. Deben tener unos límites claros y unos atributos y un comportamiento comunes para todas las instancias. Debemos eliminar las clases demasiado generales, normalmente creando otras más específicas. Las clases genéricas se incluirán más adelante si es necesario, al observar atributos u operaciones comunes a varias clases.

- **Atributos**. Los nombres que describen propiedades de los objetos son atributos, no clases. Una de las características de un objeto es su identidad (aún cuando los valores de los atributos sean comunes), y otra muy común es su existencia independiente. Los atributos de un objeto no presentan estas dos propiedades. Si la existencia independiente de un atributo es importante, hay que modelarlo como una clase. (P. ej. podemos considerar el Despacho, como un atributo de la clase Empleado, pero esto nos impide hacer una reasignación de despachos, por lo que, si queremos implementar esta operación, deberemos considerar los despachos como objetos en lugar de como atributos).

- **Operaciones**. Una clase no puede ser una operación que se aplique sobre los objetos sin tener independencia por sí misma (p. ej. en nuestro modelo de la línea telefónica la Llamada es una operación, no una clase). Sin embargo, esto depende del dominio de aplicación: en una aplicación de facturación telefónica, la Llamada será una clase que contiene atributos tales como Origen, Destino, Fecha, Hora y Número de Pasos.

- **Roles**. El nombre de una clase debe depender de su naturaleza intrínseca y no del papel que juegue en el sistema. Podemos agrupar varias clases en una, si intrínsecamente son lo mismo, y luego distinguir estos roles en las asociaciones. (P. ej. La clase *Persona* puede jugar varios papeles (*Propietario*, *Conductor*, etc.) en relación con la clase *Vehículo*. En otros casos, una misma entidad física puede modelarse mediante varias clases distintas si los atributos y el comportamiento (y especialmente las instancias) son distintos dependiendo del papel que juegue en el sistema.

- **Objetos internos**. No debemos incluir en el la fase de análisis clases que no se corresponden con el dominio de aplicación sino que tienen relación con la implementación del sistema.

Una vez identificadas las clases debemos empezar a preparar el diccionario de datos del sistema. Cada clase figurará como una entrada en el diccionario donde se define brevemente su significado y las funciones que realiza. El diccionario de datos también contiene entradas para las asociaciones, las operaciones y los atributos, que deben ser definidos según las vamos incorporando al modelo.

Identificar asociaciones (incluyendo las de composición) entre los objetos.

Cualquier relación entre dos o más clases debe ser modelada como una asociación. No hay que incluir en las clases punteros o atributos que se refieran a objetos de otras clases. Todas estas relaciones deben modelarse como asociaciones para que quede constancia de ellas en el modelo de objetos.

Las relaciones de composición también son asociaciones, simplemente tienen unas características particulares. Hay que distinguir entre unas y otras pero no merece la pena discutir mucho sobre ello.

Una vez identificadas las asociaciones candidatas, procederemos a revisar esta lista, eliminado algunas según los siguientes criterios:

- **Asociaciones irrelevantes y de implementación**. Hay que eliminar todas las asociaciones que, a pesar de que existan en el muno real, no sean relevantes para el sistema. Lo mismo sucede con las asociaciones cuya existencia se base en la implementación de la solución. Las asociaciones del modelo de objetos deben tener existencia real y ser relevantes para la solución del problema.

- **Acciones**. Las asociaciones deben describir relaciones estructurales entre las clases. Existen otros tipos de relaciones, (por ejemplo cuando un objeto recibe otro como parámetro en un mensaje que invoca una acción), que no figuran normalmente como relaciones en el modelo de objetos.

- **Asociaciones no binarias**. La mayor parte de las relaciones entre más de dos clases pueden ser expresadas mediante asociaciones binarias o con atributos (si uno de los términos no tiene existencia propia sino siempre ligada a la asociación), pero en algunos casos no podemos hacer esto sin perder información. Hasta el momento no se a mostrado como necesaria ninguna asociación entre cuatro o más clases.

- **Asociaciones derivadas**. Hay que eliminar las asociaciones que puedan ser expresadas a partir de otras (p. ej. la asociación Abuelo). La existencia de caminos múltiples entre dos clases suele indicar que existen asociaciones derivadas que pueden ser eliminadas, pero en algunos casos los caminos múltiples son necesarios.

Una vez revisada la lista, procederemos a completar el modelo de objetos, indicando para cada asociación que lo necesite:

- **Roles**. Los roles o papeles son muy útiles cuando hay asociaciones entre la misma clase o hay varias asociaciones entre un par de clases. El rol describe el papel que juega un objeto en la relación desde el punto de vista de la otra clase. (P. ej. Persona (empleado) trabaja para (contratante) Empresa.

- **Calificadores**. Normalmente los nombres sirven para identificar los objetos dentro de un contexto. De esta forma, un atributo puede servir para identificar entre las múltiples instancias conectadas al extremo M de una asociación, reduciendo así la multiplicidad de la relación.

- **Multiplicidad**. Es necesario especificar la multiplicidad, pero no hay que obsesionarse mucho pues cambia con frecuencia a lo largo del análisis. Una vez que el modelo esté completo hay que revisar las conexiones con multiplicidad 1 (que suelen ser opcionales) y las de multiplicidad M, para ver si realmente son necesarias.

- **Ordenación y otras restricciones.**

Identificar atributos y operaciones.

El siguiente paso es identificar los atributos y operaciones de cada clase y de los enlaces que los contengan. Los atributos describen propiedades de los objetos, tales como *peso*, *color* o *edad*, pero no son objetos. Cualquier relación entre objetos debe ser modelada como una asociación, no como un atributo de los objetos relacionados. Tampoco hay que incluir como atributos de los objetos los calificadores de las asociaciones ni los atributos de los enlaces.

Con frecuencia los atributos no figuran en la especificación preliminar. Lo normal es partir de un conjunto básico de atributos para cada objeto e ir añadiendo otros según los vayamos necesitando.

Organizar y simplificar las clases mediante relaciones de generalización y especialización.

El paso siguiente es organizar las clases en jerarquías de forma que se puedan abstraer las características comunes. Esto puede hacerse en dos direcciones:

- Hacia arriba (**generalización**), abstrayendo propiedades (atributos y relaciones) comunes a un grupo de clases en una superclase que las generalice. Muchas relaciones de generalización se corresponden con clasificaciones (taxonomías) de los objetos en el mundo real.

- Hacia abajo (**especialización**), refinando las clases existentes en subclases especializadas. Estas relaciones de especialización suelen aparecer en la especificación preliminar en forma de nombres adjetivados: lámpara incandescente, lampara fluorescente, etc.

Aunque la herencia facilita la reutilización de código, hay que evitar abusar demasiado de estas relaciones. Aplicaremos la especialización sólo cuando sea necesario, es decir, cuando la distinción entre dos clases especializadas sea significativa para el problema. Esto sucederá cuando ambas clases tengan atributos o operaciones distintos.

Verificar los caminos de acceso.

A continuación hay que comprobar los caminos de acceso en el modelo de objetos. Se trata de comprobar si todas las asociaciones necesarias están presentes en el diagrama, de forma que se pueda hacer un recorrido entre los objetos relacionados. También hay que comprobar la multiplicidad: para relaciones con multiplicidad M, ¿es necesario acceder a las instancias relacionadas de forma individualizada? ¿existe algún mecanismo para hacerlo?, ¿existe un camino para contestar todas las preguntas útiles acerca del problema.? (p. ej. ¿cuáles son las casas de una persona?, ¿a quién le ha comprado esta persona cada casa?, ¿cuándo se realizó la escritura?).

Refinar el modelo iterativamente.

Es muy difícil que el modelo de objetos nos salga bien a la primera. El desarrollo de software sigue normalmente un proceso iterativo: podemos encontrar errores o mejoras en nuestro modelo de objetos, no sólo mientras lo estamos haciendo, sino al hacer el modelo funcional, el dinámico o incluso en las fases de diseño o implementación. Nuestro objetivo es encontrar los errores lo antes posible, especialmente antes de entregar el software al cliente (pérdida de imagen) y antes de implementar los programas (pérdida de tiempo en codificación y pruebas). Cuanto antes encontremos los errores más sencillo y baratos será corregirlos. Por ello hay que tener especial cuidado en la fase de análisis, revisando el modelo cuantas veces sea necesario, evitando especialmente la disparidad con los modelos dinámico y funcional.

Modelo dinámico.

El modelo dinámico muestra el comportamiento de los objetos del modelo de objetos. Podemos empezar estableciendo una lista de los eventos que afectan al sistema, y establecer una o varias trazas de eventos que muestren secuencias normales de estos eventos. Luego estableceremos las secuencias de eventos permitidas para cada clase de objetos, a partir de las cuales podemos realizar los DEs.

- **Establecer una lista de posibles eventos**. Para ello partimos, como siempre, de la especificación preliminar. Hay que identificar también el objeto que emite el evento (algunos eventos provienen del entorno del sistema), el que lo recibe, y los parámetros que pudiese tener. Podemos agrupar los eventos en clases, especialmente cuando tienen parámetros o para poder utilizarlos con diferentes grados de abstracción.

- **Eliminar de la lista de eventos las operaciones que no afecten al estado de un objeto**. Los objetos reciben mensajes indicando que deben realizar alguna de las operaciones que tienen definidas, pero muchas de estas operaciones no cambiarán el estado del objeto (el objeto se comportará igual después de realizar la operación), por tanto no consideraremos estas órdenes o peticiones como eventos.

- **Realizar varias trazas de eventos**. En cada traza mostraremos una secuencia lógica de eventos junto con los objetos que los emiten y reciben. Podemos empezar haciendo trazas de los casos más habituales de funcionamiento del sistema, luego haremos trazas para situaciones menos corrientes pero igualmente válidas y acabaremos haciendo trazas para determinar el comportamiento del sistema ante situaciones erróneas. El control de errores suele ser la parte más difícil del modelo dinámico de una sistema interactivo. Hay que

prever todas las situaciones posibles (incluyendo *timeouts* para las comunicaciones o las peticiones de datos al usuario) y permitir la corrección de cualquier entrada de datos, definiendo claramente mecanismos de vuelta atrás.

- **Construir un DE para cada clase de objetos que presente estados distintos**. Sólo consideraremos los objetos que puedan presentar varios estados, es decir, que se comporten de manera distinta según valores distintos de sus atributos. Para cada una de estas clases iremos tomando las trazas donde aparece, empezando de las más habituales a las más específicas. Cada traza se corresponderá con una secuencia de transiciones en el DE. Cada vez que reciba un evento, un objeto cambiará (posiblemente) de estado, estableciendo un valor nuevo para alguno de sus atributos. Revisaremos cada DE para ver si esta completo (si se han considerado todas las transiciones) y para determinar si los estados finales son efectivamente tales.

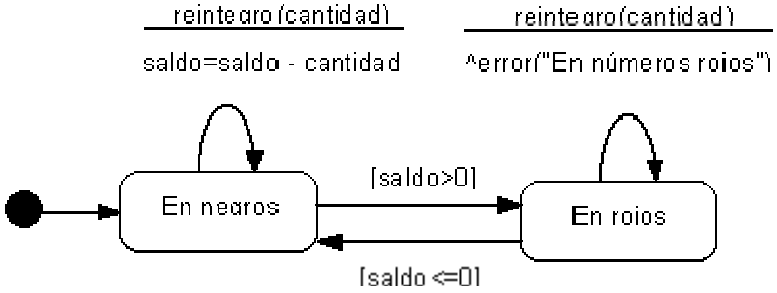
- **Verificar la consistencia de los eventos entre los diferentes DEs**. Hay que verificar la completitud y consistencia de los eventos a nivel de todo el sistema. Los eventos que emite un objeto deben ser recibidos por algún otro. Podemos completar el modelo dinámico dibujando uno o varios *diagramas de interacción de objetos,* que nos muestren las relaciones entre las distintas clases del sistema basadas en intercambio de eventos.

Modelo funcional.

El modelo funcional muestra cómo se calculan valores, independientemente de cuándo se realizan esos cálculos o de la estructura de los objetos que almacenan esos valores. Lo que muestra el modelo funcional es las relaciones de dependencia de datos y dependencia funcional.

Normalmente, la aplicación del paradigma orientado a objetos lleva a una mayor fragmentación del código que los métodos estructurados: los métodos tienen una implementación breve, basada normalmente en la llamada a otros métodos de objetos relacionados. Por este motivo, podemos describir la mayoría de los métodos como primitivas de proceso, o mediante *diagramas de interacción*, siendo raro el caso en el sea útil describir una determinada operación mediante un DFD.

El DE de cada clase contiene la mayor parte de la información necesaria para realizar el modelo funcional. Las acciones y actividades asociadas al disparo de una transición se corresponden con la especificación de la operación que figuran como evento en dicha transición. En caso de que un mismo evento figure en varias transiciones, la especificación contendrá las estructuras de control que permitan distinguir entre los diferentes estados.




```
reintegro(cantidad)
if saldo > 0
then
saldo = saldo - cantidad
else
error("En números rojos");
```

Si la especificación de una operación contiene invocaciones de operaciones (mediante envío de eventos) a otras clases, puede especificarse mediante diagramas de interacción. Por el contrario, si se trata de una operación exclusivamente interna, o si no se han realizado DE de la clase, la operación se puede especificar mediante una

primitiva de proceso. Esta descripción puede hacerse en lenguaje natural, pseudocódigo, por medio de ecuaciones matemáticas o de tablas. La desventaja de hacerlo en lenguaje natural es que la ambigüedad es mayor, pudiendo presentarse problemas a la hora de implementar y que es más difícil comprobar su consistencia. Aunque utilicemos un algoritmo (pseudocódigo) para describir un proceso, el objetivo es especificar qué hace el proceso y no cómo lo hace. La elección definitiva del algoritmo depende de consideraciones adicionales que se establecen en las fases de diseño y de implementación.

La especificación de las operaciones debe incluir también cualquier restricción que se deba establecer sobre los datos del sistema, (por ejemplo, precondiciones y postcondiciones de los procesos), así como criterios de optimización que deban tenerse en cuenta en el diseño y la optimización.

Modelo de aplicación.

Una vez realizado el modelo del dominio, donde nos habremos centrado fundamentalmente en los objetos del dominio de aplicación, podemos completarlo, convirtiéndolo en un modelo de aplicación. Para ello nos serán de gran utilidad los casos de uso desarrollados junto con la especificación inicial.

En primer lugar hemos de determinar los límites del sistema, identificando lo que es parte del mismo y lo que son actores externos. Es posible que algunos objetos que aparecen en el modelo del dominio no formen realmente parte de la aplicación. Para determinarlo, debemos observar si es necesario guardar información sobre estos objetos o si vamos a implementar las operaciones que aparecen en ellos.

Por otro lado, el examen de los casos de uso nos permitirá determinar cuál es el acceso necesario para cada clase del dominio. Se pueden construir una o más *vistas* de cada clase del dominio para proporcionar estas distintas formas de acceso. Estas vistas deben añadirse al modelo de objetos, definiendo la traducción entre objetos del dominio y *vistas*.

Debemos identificar cuales son los eventos que se intercambian entre los actores y el sistema. Los periodos de tiempo entre eventos se definirán como estados. Incluiremos objetos controladores para controlar la secuencia de eventos en cada vista. Esto nos permitirá separar la relación estática entre objetos del dominio y vistas de la dinámica de las vistas.

A partir de los casos de uso también podremos determinar cuáles serán los comandos del sistema. Estos comandos serán peticiones que realizan los actores para que el sistema realice una determinada acción. Los comandos del sistema deben añadirse al modelo funcional como operaciones. Si no es posible asignar estas operaciones a ninguna de las clases existentes, la asignaremos a un controlador.

Por último debemos determinar las interfaces externas del sistema, incluyendo las interfaces con dispositivos externos, otros sistemas, etc. Es conveniente rodear los objetos de la aplicación de objetos interfaz que los aíslen y eviten dependencias de la aplicación respecto a estos sistemas externos.

Refinar el modelo.

Una vez que hemos desarrollado el modelo del sistema, nos quedan aún una serie de cosas por hacer:

Completar la lista de operaciones.

La primera de ellas es **completar la lista de operaciones**, dado que normalmente no es posible hacerlo al desarrollar el modelo de objetos, sino que van apareciendo a lo largo del análisis. Podemos clasificar las operaciones de los objetos en los siguientes grupos:

- **Operaciones implícitas sobre objetos**. Normalmente, cualquier sistema debe ser capaz de crear, destruir y copiar instancias de las clases que define. Algunas de estas operaciones están, directa o indirectamente, descritas en la especificación preliminar, pero la mayor parte de las veces están implícitas. Los LOO suelen incluir primitivas de creación, destrucción y copia de las instancias de una clase.

- **Operaciones implícitas sobre atributos**. Existe otro grupo de operaciones implícitas: las encargadas de consultar o asignar el valor de cada uno de los atributos de un objeto. Así mismo, deben existir operaciones que recorran los enlaces que una instancia tenga con otras. En la descripciones de primitivas de proceso podemos utilizar la notación siguiente, para referirnos a estas operaciones:

clase.atributo	operación de consulta que devuelve el valor del atributo.
clase.atributo := valor	operación de asignación.
clase.asociación	atributo que apunta al objeto asociado.
clase.asociación[i]	id. para asociaciones múltiples y ordenadas.
clase.asociación[calificador]	id. para asociaciones calificadas.

- **Operaciones invocadas por eventos**. La recepción de un evento trae consigo la ejecución de una serie de operaciones (acciones y actividades) en el objeto que recibe el mensaje. Es habitual incluir en la clase receptora operaciones con mismo nombre que el evento que las desencadena, en las que se incluyen estas acciones y actividades.

- **Operaciones invocadas por acciones y actividades**. Por el contrario, en algunos casos puede ser conveniente representar las propias acciones y/o actividades como operaciones de las clases. Se tratará en estos casos de operaciones auxiliares que se invocan a la recepción de los eventos correspondientes.

Una vez que hemos completado la lista de operaciones de cada clase, debemos comprobar estas listas con vistas a simplificar operaciones redundantes o a establecer relaciones de generalización entre clases que tengan una serie de operaciones comunes.

Comprobar la consistencia entre los modelos.

El proceso de análisis no es una actividad secuencial, sino que según vamos desarrollando un modelo surgen modificaciones de los anteriores. Una vez que tenemos los tres modelos hay que comprobar la consistencia, para ver si alguna de estas modificaciones se nos ha pasado por alto. Para comprobar la consistencia nos basaremos en las relaciones entre los modelos de las que ya hemos hablado.

Aunque el primer modelo que realicemos sea consistente, rara vez será correcto. Demos revisarlo para ver si se nos ha olvidado algo de lo establecido en la especificación preliminar o si notamos algo extraño.

Comprobar el modelo con el cliente.

Una vez que nosotros hemos dado el visto bueno al análisis, el cliente debe hacer lo mismo. En primer lugar debemos mostrarle los resultados del análisis para ver si son acordes con sus necesidades, o si hemos cometido algún error o se nos ha pasado por alto algún detalle. También debemos discutir con el las modificaciones que se hayan realizado con respecto a lo establecido en la especificación preliminar, para ver si las suposiciones que hemos hecho son correctas.

Deficiencias del análisis estructurado.

- **Descomposición funcional**. Requiere traducir el sistema a una serie de funciones y subfunciones.

- **Flujo de datos**. Es un modelo informático, no nuestra forma habitual de pensar.

- **Modelo de datos**. La relación con el modelo de procesos es débil. Pueden dar visiones muy distintas del mismo problema.

Ventajas del análisis orientado a objetos.

- **Dominio del problema**. Representa el sistema en términos del mundo real, en vez de en términos informáticos.
- **Comunicación**. Usa conceptos más simples. La comunicación con el cliente es más fácil.
- **Consistencia**. Reduce la distancia entre el modelo de procesos y el de datos.
- **Expresión de características comunes**. Evita la duplicación de características comunes a varios elementos.
- **Resistencia al cambio**. Al ser más próximo al sistema real es más estable frente a cambios en los requisitos.
- **Reutilización**. Favorece la reutilización, tanto interna como externa.

Características del enfoque orientado a objetos.

Identidad.

- Los elementos del dominio del problema se organizan en entidades discretas y distinguibles llamadas **objetos**.
- Los objetos encapsulan **atributos** (datos) y **operaciones**.
- Cada objeto tiene **identidad propia**, aunque los valores de sus atributos coincidan con los de otro.

Clasificación.

- Los objetos con propiedades comunes se agrupan en **clases**.
- Las clases son **abstracciones** de características comunes a una serie de objetos.
- Las clases son **arbitrarias**: dependen del dominio del problema.
- Cada uno de los objetos agrupados en una clase se llama **instancia**.
- Las instancias de una clase comparten atributos, operaciones y comportamiento pero tienen valores distintos en los atributos.

Polimorfismo.

- Una misma operación puede realizarse de formas distintas en clases distintas. La **semántica** es común pero la **implementación** varía en cada clase.
- La implementación de una operación en una clase se denomina **método**.

Herencia.

- Es una **relación jerárquica entre clases** con características comunes.
- La **clase padre** define características comunes a todas las hijas.
- Cada **clase hija** hereda las características del padre y las amplia o refina.

Elementos del modelo de objetos

Instancias. Cada uno de los objetos individuales.

Clases. Abstracción de objetos con propiedades comunes.

Atributos. Datos que caracterizan las instancias de una clase**.**

Operaciones. Funciones que pueden realizar las instancias**.**

Relaciones. Se establecen entre clases.

Asociación. Relación de uso en general.

Multiplicidad. Número de instancias que intervienen en la relación.

Atributos. Algunos atributos pueden depender de la asociación.

Calificación. Limita la multiplicidad de las asociaciones.

Roles. Indican los papeles de las clases en las relaciones.

Restricciones y ordenación.

Composición. Relaciones todo/parte.

Generalización. Relaciones padre/hijo.

Redefinición. Modificación de las propiedades heredadas.

Enlaces. Instancias de una relación. Relaciona instancias.

Modelo de objetos: consejos prácticos.

- No lanzarse a dibujar clases y asociaciones sin sentido.
- Intentar que el modelo sea simple, evitando complicaciones innecesarias.
- Los nombres de objetos, asociaciones, atributos y operaciones deben ser significativos.
- No incluir en los objetos punteros a otros objetos.
- Utilizar, si es posible, asociaciones binarias.
- Dejar la definición de la multiplicidad para cuando se tenga un mejor conocimiento del problema.
- No incluir los atributos de las asociaciones en las clases.
- Utilizar preferentemente asociaciones cualificadas en vez de ternarias o con atributos.
- Evitar las jerarquías de composición o generalización de muchos niveles.
- Revisar el modelo hasta que sea satisfactorio.
- Documentar el modelo.

- Utilizar sólo los elementos necesarios.

Modelo dinámico.

- **Estado de un objeto**. Conjunto de valores de los atributos y enlaces que tiene un objeto en un momento determinado.
- **Diagrama de transición de estados**. Diagrama que muestra la secuencia de estados, eventos y acciones de un objeto.
- **Modelo dinámico de un sistema**. Conjunto de DEs de las clase de objetos del sistema.
- **Eventos**. Señales mediante las que se comunican los objetos. Pueden ser señales lógicas o contener atributos.
- **Estado**. Abstracción de los estados de un objeto en el que éste presenta determinado comportamiento. Engloba un conjunto de valores de los atributos y enlaces (un conjunto de estados del objeto).
- **Trazas de eventos**. Diagrama que muestra la secuencia de envíos y recepciones de eventos entre varios objetos.
- **Guarda**. Condición de datos que debe cumplir un objeto para que se dispare una transición al recibir un determinado evento. (*Notación*: [Guarda]).
- **Actividad**. Operación que necesita un tiempo para ejecutarse. Se representa en los estados. (*Notación*: Hacer: Actividad)
- **Acción**. Operación que se realiza (idealmente) de forma instantánea. Se representa en las transiciones. (*Notación*: Evento [Guarda] / Acción).

Modelo dinámico: consejos prácticos.

- No todas las clases necesitan un DE.
- Utilizar trazas de eventos como ayuda para construir el DE.
- Para definir estados, sólo hay que tener en cuenta los atributos que influyen en el comportamiento.
- Decidir la granularidad de estados y eventos según las necesidades del modelo
- Distinguir entre eventos y guardas.
- Distinguir entre actividades y acciones.
- Utilizar diagramas anidados cuando una transición se aplique a varios estados.
- Realizar un DE compuesto si:
    - ¡ Existen relaciones de composición entre los objetos.
    - ¡ Existe concurrencia interna en el objeto.
    - ¡ Podemos dividir el comportamiento en facetas independientes.

¡ Existen relaciones de herencia entre los objetos.

- En caso de herencia de entre clases, el DE de la subclase **no puede añadir estados o transiciones al DE padre**.
- Comprobar la consistencia de los eventos que se generan en un DE y se utilizan en otros.

Modelo de objetos I

Pasos a seguir.

- Identificar objetos y clases.
- Preparar un diccionario de datos.
- Identificar asociaciones (incluyendo las de composición) entre los objetos.
- Identificar atributos y operaciones.
- Organizar y simplificar las clases mediante relaciones de generalización y especialización.
- Verificar los caminos de acceso.
- Refinar el modelo iterativamente.

Eliminar de la lista de clases candidatas:

- Clases redundantes.
- Clases irrelevantes.
- Clases demasiado generales.
- Atributos.
- Operaciones.
- Roles.
- Construcciones de implementación.

Modelo de objetos II

Eliminar de la lista de asociaciones candidatas:

- Asociaciones irrelevantes y de implementación.
- Acciones.
- Asociaciones no binarias.
- Asociaciones derivadas.

Completar las asociaciones indicando:

- Roles.

- Calificadores.
- Multiplicidad.
- Ordenación.

Modelo dinámico. Pasos a seguir:

- Establecer una lista de posibles eventos.
- Eliminar de la lista de eventos las operaciones que no afecten al estado de un objeto.
- Realizar varias trazas de eventos.
- Construir un DE para cada clase de objetos que presente estados distintos, incorporando una a una las trazas de eventos.
- Verificar la consistencia de los eventos entre los diferentes DEs: construir un diagrama de flujo de eventos.

Modelo funcional. Pasos a seguir.

- Identificar datos de entrada y salida.
- Hacer DFDs para mostrar la dependencia funcional.
- Describir las primitivas de proceso.

4. Herramientas CASE

Hoy en día, muchas empresas se han extendido a la adquisición de herramientas CASE (Ingeniería Asistida por Computadora), con el fin de automatizar los aspectos clave de todo el proceso de desarrollo de un sistema, desde el principio hasta el final e incrementar su posición en el mercado competitivo, pero obteniendo algunas veces elevados costos en la adquisición de la herramienta y costos de entrenamiento de personal así como la falta de adaptación de la herramienta a la arquitectura de la información y a las metodologías de desarrollo utilizadas por la organización. Por otra parte, algunas herramientas CASE no ofrecen o evalúan soluciones potenciales para los problemas relacionados con sistemas o virtualmente no llevan a cabo ningún análisis de los requerimientos de la aplicación. Sin embargo, él CASE proporciona un conjunto de herramientas semiautomatizadas y automatizadas que están desarrollando una cultura de ingeniería nueva para muchas empresas. Uno de los objetivos más importante del CASE (a largo plazo) es conseguir la generación automática de programas desde una especificación a nivel de diseño. Ahora bien, con la aparición de las redes de ordenadores en empresas y universidades ha surgido en el mundo de la informática la tecnología cliente / servidor. Son muchas de las organizaciones que ya cuentan con un número considerable de aplicaciones cliente / servidor en operación: Servidores de Bases de Datos y Manejadores de Objetos Distribuidos. Cliente / servidor es una tecnología de bajo costo que proporciona recursos compartidos, escalabilidad, integridad, encapsulamiento de servicios, etc. Pero al igual que toda tecnología, el desarrollo de aplicaciones cliente / servidor requiere que la persona tenga conocimientos, experiencia y habilidades en procesamiento de transacciones, diseño de base de datos, redes de ordenadores y diseño gráfica de interfase. El objeto de estudio está centrado en determinar ¿cuáles son las influencias de las herramientas CASE en las empresas desarrolladoras de sistemas de información cliente / servidor? Y ¿cuáles son las tendencias actuales de las empresas fabricantes de sistemas cliente / servidor?. Como soporte, se planteará un marco teórico que explicará la filosofía cliente / servidor, herramientas CASE y sistemas de información, y posteriormente se procederá a responder a las preguntas mencionadas anteriormente.

- ***Tipos (uppercase, lowercase)***

Clasificación de las herramientas case

Como ya hemos comentado en los apartados precedentes CASE es una combinación de herramientas software (aplicaciones) y de metodologías de desarrollo :

- Ø Las herramientas permiten automatizar el proceso de desarrollo del software.
- Ø Las metodologías definen los procesos automatizar.

Una primera clasificación del CASE es considerando su amplitud :

- Ø TOOLKIT: es una colección de herramientas integradas que permiten automatizar un conjunto de tareas de algunas de las fases del ciclo de vida del sistema informático: Planificación estratégica, Análisis, Diseño, Generación de programas.
- Ø WORKBENCH: Son conjuntos integrados de herramientas que dan soporte a la automatización del proceso completo de desarrollo del sistema informático. Permiten cubrir el ciclo de vida completo. El producto final aportado por ellas es un sistema en código ejecutable y su documentación.

Una segunda clasificación es teniendo en cuenta las fases (y/o tareas) del ciclo de vida que automatizan:

- Ø UPPER CASE: Planificación estratégica, Requerimientos de Desarrollo Funcional de Planes Corporativos.
- Ø MIDDLE CASE: Análisis y Diseño.
- Ø LOWER CASE: Generación de código, test e implantación

- ***Componentes***

¿Qué es case?

"CASE es la automatización del software"
(Carma MacClure)

"CASE es una filosofía que se orienta a la mejor comprensión de los modelos de empresa, sus actividades y el desarrollo de los sistemas de información. Esta filosofía involucra además el uso de programas que permiten:

Construir los modelos que describen la empresa,
Describir el medio en el que se realizan las actividades,
Llevar a cabo la planificación,
El desarrollo del Sistema Informático, desde la planificación, pasando por el análisis y diseño de sistemas, hasta la generación del código de los programas y la documentación."

(Michael Lucas Gibson)

"La creación de sistemas software utilizando técnicas de diseño y metodologías de desarrollo bien definidas, soportadas por herramientas automatizadas operativas en el ordenador"

Objetivos del case

A. Aumentar la productividad de las áreas de desarrollo y mantenimiento de los sistemas informáticos.
B. Mejorar la calidad del software desarrollado.
C. Reducir tiempos y costes de desarrollo y mantenimiento del software.
D. Mejorar la gestión y dominio sobre el proyecto en cuanto a su planificación, ejecución y control.
E. Mejorar el archivo de datos (enciclopedia) de conocimientos (know-how) y sus facilidades de uso, reduciendo la dependencia de analistas y programadores.
F. Automatizar :

- Ø El desarrollo del software
- Ø La documentación
- Ø La generación del código
- Ø El chequeo de errores
- Ø La gestión del proyecto

G. Permitir:

- Ø La reutilización (reusabilidad) del software
- Ø La portabilidad del software
- Ø La estandarización de la documentación

H. Integrar las fases de desarrollo (ingeniería del software) con las herramientas CASE
I. Facilitar la utilización de las distintas metodologías que desarrollan la propia ingeniería del software.

Enciclopedia (repository)

En el contexto CASE se entiende por enciclopedia a la base de datos que contiene todas las informaciones relacionadas con las especificaciones, análisis y diseño del software. En está base de datos se incluyen las informaciones de :

A. DATOS: Elementos atributos (campos), asociaciones (relaciones), entidades (registros), almacenes de datos, estructuras, etc.
B. PROCESOS: Procesos, Funciones, módulos, etc.

C. GRAFICOS: DFD (Diagrama de flujo de datos), DER (Diagrama Entidad Relación) DFD (Diagrama de Descomposición Funcional), ED (Diagrama de Estructura), Diagrama de Clases, etc.
D. REGLAS: de Gestión, de métodos, etc.

El case en el "ciclo de vida del sistema" desarrollado en el curso de análisis y diseño

Ø Ciclo de Vida :

Etapas en un proyecto case

Para llevar a cabo con éxito el proyecto de introducción del CASE en el Área de Desarrollo, en el CEDS recomendamos que como mínimo se tengan en cuenta cinco etapas:

ETAPA 1 : Descripción de Objetivos - Grupo de Trabajo – Planificación provisional del proyecto.
ETAPA 2 : Análisis del Área de Desarrollo
ETAPA 3 : Selección de Metodología y Herramientas CASE
ETAPA 4 : Aplicación en Escenarios y Evaluación (es muy importante que el proyecto de evaluación NO sea crítico y su tamaño pequeño)
ETAPA 5 : Extensión de la Metodología y CASE en la Organización

Causas por las que fracasan algunos proyectos case

No siempre han tenido éxito los proyectos de introducción del CASE. Bien es cierto que debido a que los nuevos programas de formación de Analistas ya tienen en cuenta tanto la Metodología como el uso y prácticas con sistemas CASE, están permitiendo reducir los riegos de fracaso. No obstante en muchas organizaciones actuales no se dispone de Analistas formados, ni de experiencias CASE. Son estas organizaciones las que deben poner especial atención en las causas mas frecuentes por las que puede fracasar el proyecto :

- Ø No se tienen en cuenta las tres primeras etapas,
- Ø No se concreta ninguna Metodología,
- Ø El proyecto de evaluación es demasiado ambicioso ó crítico,
- Ø En la etapa quinta no se lleva a cabo la Formación que se precisa,
- Ø Los Usuarios (Área de Desarrollo), no están motivados.

Un proyecto de introducción de CASE es siempre "un proyecto estratégico" para el Área de Desarrollo y como tal "No tiene vuelta atrás". Cuando la decisión ya ha sido tomada "siga con pasos firmes todas las etapas" teniendo muy en cuenta que "Los tiempos y esfuerzos para cubrirlas dependerán de las personas que integran el Área de Desarrollo". En organizaciones muy preparadas, su introducción ha requerido un año.

Características deseables de una case

Una herramienta CASE cliente / servidor provee modelo de datos, generación de código, registro del ciclo de vida de los proyectos, múltiples repositorios de usuarios, comunicación entre distintos ingenieros. Las principales herramientas son KnowledgeWare's Application Development Workbench, TI's Information Engineering Facility (IEF), and Andersen Consulting's Foundation for Cooperative Processing. Por otra parte, una herramienta CASE Cliente / servidor debe ofrecer:

- Ø Proporcionar topologías de aplicación flexibles. La herramienta debe proporcionar facilidades de construcción que permita separar la aplicación (en muchos puntos diferentes) entre el cliente, el servidor y más importante, entre servidores.
- Ø Proporcionar aplicaciones portátiles. La herramienta debe generar código para Windows, OS/ 2, Macintosh, Unix y todas las plataformas de servidores conocidas. Debe ser capaz, a tiempo de corrida, desplegar la versión correcta del código en la máquina apropiada.
- Ø Control de Versión. La herramienta debe reconocer las versiones de códigos que se ejecutan en los clientes y servidores, y asegurarse que sean consistentes. También, la herramienta debe ser capaz de

controlar un gran número de tipos de objetos incluyendo texto, gráficos, mapas de bits, documentos complejos y objetos únicos, tales como definiciones de pantallas y de informes, archivos de objetos y datos de prueba y resultados. Debe mantener versiones de objetos con niveles arbitrarios de granularidad; por ejemplo, una única definición de datos o una agrupación de módulos.
Ø Crear código compilado en el servidor. La herramienta debe ser capaz de compilar automáticamente código 4GL en el servidor para obtener el máximo performance.
Ø Trabajar con una variedad de administradores de recurso. La herramienta debe adaptarse ella misma a los administradores de recurso que existen en varios servidores de la red; su interacción con los administradores de recurso debería ser negociable a tiempo de ejecución.
Ø Trabajar con una variedad de software intermedios. La herramienta debe adaptar sus comunicaciones cliente / servidor al software intermedio existente. Como mínimo la herramienta debería ajustar los temporizadores basándose en, si el tráfico se está moviendo en una LAN o WAN.
Ø Soporte multiusuarios. La herramienta debe permitir que varios diseñadores trabajen en una aplicación simultáneamente. Debe gestionarse los accesos concurrentes a la base de datos por diferentes usuarios, mediante el arbitrio y bloqueos de accesos a nivel de archivo o de registro.
Ø Seguridad. La herramienta debe proporcionar mecanismos para controlar el acceso y las modificaciones a los que contiene. La herramienta debe, al menos, mantener contraseñas y permisos de acceso en distintos niveles para cada usuario. También debe facilitar la realización automática de copias de seguridad y recuperaciones de las mismas, así como el almacenamiento de grupos de información determinados, por ejemplo, por proyecto o aplicaciones.
Ø Desarrollo en equipo, repositorio de librerías compartidas. Debe permitir que grupos de programadores trabajen en un proyecto común; debe proveer facilidades de check-in/ check-out registrar formas, widgets, controles, campos, objetos de negocio, DLL, etc.; debe proporcionar un mecanismo para compartir las librerías entre distintos realizadores y múltiples herramientas; Gestiona y controla el acceso multiusuario a los datos y bloquea los objetos para evitar que se pierdan modificaciones inadvertidamente cuando se realizan simultáneamente.

III. Administración del desarrollo y calidad del sistema

1. Administración de proyectos

- ***Estimación de tiempo y costo de cada actividad***

Duración estimada de cada actividad

Estimar cuanto durará cada actividad desde el momento que se inicie hasta que se termine este tiempo o duración estimada tiene que ser el tiempo total transcurrido –el tiempo que se haga el trabajo más cualquier tiempo de espera relacionado.

Cálculo de tiempos.

Una vez que se tiene la duración para cada actividad es importante que se determine si las actividades a realizarse serán realmente terminadas en el tiempo posible

Tiempos de inicio y terminación más tempranos.

Conociendo la duración estimada para cada actividad en la red y utilizando como referencia el tiempo de inicio del proyecto se pueden calcular los tiempos siguientes para cada actividad:

a) Tiempo de inicio más temprano TIT (ES Earliest Start Time). Es lo más pronto que se pude iniciar una actividad en particular, y calculada sobre la base de tiempo de inicio estimado del proyecto y de las duraciones estimadas de las actividades precedentes.
b) Tiempo de terminación más temprano TTT (EF Earliest Start Time). Es lo más pronto en que se puede terminar una actividad en particular y se calcula sumando la duración estimada de la actividad al tiempo de inicio más temprano de la misma.

EF = ES + Duración estimada

Los tiempos ES y EF se determinan calculando hacia delante, es decir, trabajando a través del diagrama de red desde el inicio del proyecto hasta el final del mismo. Al hacer esto se debe seguir una regla:

El tiempo de inicio más temprano de una actividad particular es el mayor tiempo de terminación más temprano de todas las precedentes inmediatas a esta actividad.

Tiempos de inicio y terminación más tardíos

Con los tiempos de duración de cada actividad se podrá calcular los tiempos de inicio y terminación tardíos.

a) Tiempo de terminación más tardíos (LF). Es lo más tarde que se puede completar una actividad en particular para que todo el proyecto se termine en la fecha acordada.
b) Tiempo de inicio más tardío (LS). Es la fecha más tardía en que se puede iniciar una actividad en particular para que todo el proyecto se complete en su fecha de terminación requerida.

LS = LF – Duración estimada

Los tiempos LS y LF se determinan calculando hacia atrás, es decir, trabajando el diagrama de red hasta el inicio del mismo, atendiendo la siguiente regla:

El tiempo de terminación más tardío para una actividad en particular es el menor de los tiempos de inicio más tardíos de todas las actividades que surgen directamente de esa en particular.

Estimación de costos del proyecto

Los costos se estiman sobre el desarrollo de la propuesta por el contratista o el equipo de trabajo. La sección de costos de una propuesta probablemente consista en tablas de gastos estimados como son:

1. Mano de obra
2. Materiales
3. Alquiler de equipo o instalaciones
4. Subcontratistas o asesores

Elaboración del presupuesto del proyecto

a) El costo estimado del proyecto se asigna a los diversos paquetes de trabajo en la estructura de división del trabajo.
b) El importe de cada paquete se distribuye a los largo de su duración, con el fin de determinar cuanto del presupuesto se gastó en cualquier momento.

- ***Planificación temporal de proyectos (gráficas de Gantt y ruta crítica)***

Ruta crítica:

Es la ruta más larga que se tiene para concluir una actividad o proyecto desde que inicia hasta que llega a su actividad final. Al trayecto más largo en el diagrama de red global se le denomina “ruta crítica”. Esta está determinada por los valores negativos que indicarán el trayecto más largo. Cuando existan valores positivos, en el cálculo de la holgura total son rutas no críticas. Los valores de cero o negativos en la holgura total se les denomina ruta crítica.

2. Aseguramiento de la calidad

- ***Concepto de calidad***

La calidad en los proyectos significa básicamente cumplir con los requerimientos del cliente. El hecho de incorporar calidad en los productos y servicios del proyecto requiere comprender los requerimientos del proyecto y luego orientar el trabajo hacia el logro del resultado final buscado. Calidad no significa exceder los requerimientos del cliente, sino satisfacerlos.

La calidad del software es difícil de medir, pero tenemos algunas pautas, algunos indicadores que nos ayudan a diferenciar los productos de calidad de los que carecen de ella:

• El acercamiento a cero defectos
• El cumplimiento de los requisitos intrínsecos y expresos
• La satisfacción del cliente

Sobre todo la satisfacción del cliente. Ya se sabe que un suministrador puede engañar a todos alguna vez o a alguno muchas veces, pero no puede engañar a muchos durante largo tiempo. El cliente siempre tiene razón y para eso están las encuestas de satisfacción. El argumento de la calidad es exhibido por las empresas como un factor diferenciador, como clave de sus procesos de negocio y como eslogan de competitividad empresarial. De hecho, la exigencia cada vez mayor por parte del mercado de la certificación ISO 9000 o el interés creciente por los modelos de calidad de gestión empresarial de tipo (EFQM) son indicadores de la percepción de la calidad como un elemento cada vez más necesario.

¿Qué debemos de entender por calidad del software?

Según la definición que encontramos en Pressman: “ Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente.”

En esta definición podemos destacar qué se entiende por calidad, el cumplimiento de los requerimientos que se han establecido (normalmente por el usuario o el cliente) y las "características implícitas" que debe cumplir todo software hecho profesionalmente aparte de su realización según unos determinados estándares (modelo de referencia). Es decir, que además de cumplir con las especificaciones que haya definido el cliente o el usuario, debe cumplir con otras características que se dan por sobreentendido que están dentro del "saber hacer" de un buen profesional y que no están específicamente documentadas. En muchas ocasiones, esta circunstancia no se da, y algunos desarrolladores de cuestionable profesionalidad se protegen tras la frase: "de eso no se dieron especificaciones", para ocultar una falta de previsión o una en el software. Carencia de habilidad para obtener del usuario en las entrevistas la información necesaria para completar y complementar los requerimientos funcionales.

Quien esté a cargo de la producción del software, ya sea un proveedor externo o el área de desarrollo de sistemas de nuestra organización, deberá contar con un Plan General de Calidad en el que estarán las especificaciones para poder definir cada uno de los Planes específicos de nuestros desarrollos en función de los atributos de Calidad que deseamos implantar en el software.

- ***Estándares***

Dado que el Plan se basa en normas de referencia, es importante mencionar algunos de ellos:
• Familia de Normas ISO 9000 y en especial la ISO 9001 y la ISO 9000-3.2: 1996 Quality Management and Quality Assurance Standards
• ISO 8402: 1994
• IEEE 730/1984, Standard for Software Quality Assurance Plans
• IEEE Std 1028: 1989, IEEE Standard for Software Reviews and Audits
• CMM. Capability Maturity Model
• ISO/IEC JTC1 15504. SPICE. Software Process Improvement and Capability Determination.
• Modelo de EFQM. Modelo de la Fundación Europea de Gestión de Calidad

- ***Factores de calidad (corrección, confiabilidad, eficiencia, facilidad de uso, de prueba y de mantenimiento, adaptabilidad y flexibilidad, portabilidad, reuso, completad, facilidad de auditoria)***

El modelo de McCall organiza los factores en tres ejes o puntos de vista desde los cuales el usuario puede contemplar la calidad de un producto:
- Operación del producto
- Revisión del producto
- Transición del producto

Los factores de McCall se definen como sigue:
1. **Corrección**: Hasta qué punto un programa cumple sus especificaciones y satisface los objetivos del usuario. Por ejemplo, si un programa debe ser capaz de sumar dos números y en lugar de sumar los multiplica, es un programa incorrecto. Es quizás el factor más importante, aunque puede no servir de nada
sin los demás factores.
2. **Fiabilidad**: Hasta qué punto se puede confiar en el funcionamiento sin errores del programa. Por ejemplo, si el programa anterior suma dos números, pero en un 25% de los casos el resultado que da no es correcto, es poco fiable.
3. **Eficiencia**: Cantidad de código y de recursos informáticos (CPU, memoria) que precisa un programa para desempeñar su función. Un programa que suma dos números y necesita 2 MB de memoria para funcionar, o que tarda 2 horas en dar una respuesta, es poco eficiente.
4. **Integridad**: Hasta qué punto se controlan los accesos ilegales a programas o datos. Un programa que permite el acceso de personas no autorizadas a ciertos datos es poco íntegro.
5. **Facilidad de uso:** El coste y esfuerzo de aprender a manejar un producto, preparar la entrada de datos e interpretar la salida del mismo.
6. **Facilidad de mantenimiento**: El coste de localizar y corregir defectos en un programa que aparecen durante su funcionamiento.
7. **Facilidad de prueba**: El coste de probar un programa para comprobar que satisface sus requisitos. Por ejemplo, si un programa requiere desarrollar una simulación completa de un sistema para poder probar que funciona bien, es un programa difícil de probar.

8. **Flexibilidad**: El coste de modificación del producto cuando cambian sus especificaciones.
9. **Portabilidad** (o Transportabilidad): El coste de transportar o migrar un producto de una configuración hardware o entorno operativo a otro.
10. **Facilidad de Reutilización**: Hasta qué punto se puede transferir un modulo o programa del presente sistema a otra aplicación, y con qué esfuerzo.
11. **Interoperabilidad**: El coste y esfuerzo necesario para hacer que el software pueda operar conjuntamente con otros sistemas o aplicaciones software externos.

Cada uno de estos factores se descompone, a su vez, en criterios. En el modelo de McCall se definen un total de 23 criterios, con el siguiente significado:
1. **Facilidad de operación**: Atributos del software que determinan la facilidad de operación del software.
2. **Facilidad de comunicación**: Atributos del software que proporcionan al usuario entradas y salidas fácilmente asimilables.
3. **Facilidad de aprendizaje**: Atributos del software que facilitan la familiarización inicial del usuario con el software y la transición desde el modo actual de operación.
4. **Control de accesos**: Atributos del software que proporcionan control de acceso al software y los datos que maneja.
5. **Facilidad de auditoría**: Atributos del software que facilitan el registro y la auditoría de los accesos al software.
6. **Eficiencia en ejecución**: Atributos del software que minimizan el tiempo de procesamiento.
7. **Eficiencia en almacenamiento**: Atributos del software que minimizan el espacio de almacenamiento necesario.
8. **Precisión**: Atributos del software que proporcionan el grado de precisión requerido en los cálculos y los resultados.
9. **Consistencia**: Atributos del software que proporcionan uniformidad en las técnicas y notaciones de diseño e implementación utilizadas.
10. **Tolerancia a fallos**: Atributos del software que posibilitan la continuidad del funcionamiento bajo condiciones no usuales.
11. **Modularidad**: Atributos del software que proporcionan una estructura de módulos altamente independientes.
12. **Simplicidad**: Atributos del software que posibilitan la implementación de funciones de la forma más comprensible posible.
13. **Completitud**: Atributos del software que proporcionan la implementación completa de todas las funciones requeridas.
14. **Trazabilidad** (Rastreabilidad): Atributos del software que proporcionan una traza desde los requisitos a la implementación con respecto a un entorno operativo concreto.
15. **Auto descripción**: Atributos del software que proporcionan explicaciones sobre la implementación de las funciones.
16. **Capacidad de expansión**: Atributos del software que posibilitan la expansión del software en cuanto a capacidades funcionales y datos.
17. **Generalidad**: Atributos del software que proporcionan amplitud a las funciones implementadas.
18. **Instrumentación**: Atributos del software que posibilitan la observación del comportamiento del software durante su ejecución (para facilitar las mediciones del uso o la identificación de errores).
19. **Independencia entre sistema y software**: Atributos del software que determinan su independencia del entorno operativo.
20. **Independencia del hardware**: Atributos del software que determinan su independencia del hardware.
21. **Compatibilidad de comunicaciones**: Atributos del software que posibilitan el uso de protocolos de comunicación e interfaces estándar.
22. **Compatibilidad de datos**: Atributos del software que posibilitan el uso representaciones de datos estándar.
23. **Concisión**: Atributos del software que posibilitan la implementación de una función con la menor cantidad de código posible.

IV. Industria del software

1. Proceso de desarrollo

- ***Pruebas betas***

El primer número de una serie, por ejemplo 1.0.1, representa la versión principal del programa. El segundo número significa una actualización en general, que sólo agrega algunos detalles nuevos.

De esta manera, cuando usted compra un programa con número de versión 3.0, obtiene una versión principal. Si opta por un producto con número de versión 3.1, adquiere entonces la versión principal 3.0, pero con algunas mejoras incorporadas.

Si un número de versión consta de tres dígitos, el tercero representa un "parche", que corrige los fallos descubiertos después de que el programa salió al mercado. De este modo, si su software tiene una versión 3.1.1, usted adquirió una versión mejorada de la versión 3.0, pero con algunas correcciones y parches adicionales. Por lo regular, los parches pueden descargarse sin costo desde Internet.

Las versiones

Pero aun antes de salir al mercado o de estar disponible en forma gratuita, un programa pasa por un proceso previo de pruebas, que evalúa su factibilidad, funcionamiento y facilidad de uso. Las principales son las siguientes:
•Pruebas Alfa. Sirven para probar una versión pre-lanzada (y potencialmente no fiable) de un programa, poniéndolo a disposición de un núcleo de usuarios muy seleccionado. Una prueba "alfa" implica una prueba local delimitada, que no está al alcance de los usuarios comunes. Por ejemplo, la próxima versión del sistema operativo de Windows, denominada longhorn, se encuentra en pruebas alfa (www.winsupersite.com/reviews/longhorn_alpha.asp).

•Pruebas Beta. Consisten en una serie de pruebas para un programa informático antes de su lanzamiento comercial. Se realizan después de las pruebas alfa y, regularmente, implican enviar el software a ciertos sitios de prueba disponibles en Internet, donde se ofrece la descarga del programa de prueba sin costo alguno.

En términos generales, se dice que un programa está en pruebas beta cuando sus desarrolladores lo han finalizado, pero están buscando ayuda de los usuarios para encontrar fallos y problemas de funcionamiento antes de lanzar la versión comercial final. Office 2003, por ejemplo, se encuentra en pruebas beta (www.microsoft.com/office/preview).

•Pruebas RC (Release Candidate: lanzamiento de candidato). Por último, después de las pruebas alfa y beta, se puede lanzar una o más versiones conocidas como Release Candidate, con el propósito de ponderar la opinión final del usuario antes del lanzamiento definitivo. Mozilla, un explorador alternativo de Internet, tiene varias versiones RC (www.mozilla.org/releases/ mozilla1.4rc1).

Por su parte, algunas empresas como Microsoft prefieren que sus productos sean conocidos por un nombre más que por su versión, por ejemplo Windows XP. Se supone que, de este modo, los usuarios identifican más rápidamente los productos.

La empresa de las ventanas ofrece sus actualizaciones y parches mediante los denominados "service packs" (paquetes de servicio), que se componen principalmente de innumerables mejoras y parches de seguridad, con crípticos nombres como MS02-055.

El final del producto

Ocurre que un fabricante de software deja de dar respaldo a una versión de uno de sus programas luego de cierto tiempo en el mercado; es decir, ya no proporciona servicio técnico para sus productos "viejos". Esto no significa que un programa deje de ser usado, pero sí significa que no saldrán nuevas actualizaciones o parches, aunque se detecten problemas en el programa. Es, por decirlo así, la fase final o deceso de esa versión.

Para los usuarios esto significa que debe pensarse muy seriamente en cambiar de versión del programa, a efecto de mantenerse actualizado en materia de fallos y seguridad. Para las empresas, representa un medio de mantener altas sus ventas.

- ***Control de versiones***

Los cambios son un elemento medular de la vida del desarrollo de software. El trabajo efectivo requiere de una administración formal de los cambios. Cuando se cuenta con una administración de cambios del software realmente efectiva se logra que:

- Los equipos de desarrollo pueden hacer liberaciones a tiempo, en presupuesto y con una calidad predecible.
- Los líderes de proyecto conozcan en todo momento el estado y avance del desarrollo y tengan certeza en el tiempo de liberación.
- Los testers sepan cuando una nueva construcción requiere ser probada y qué mejoras o correcciones debe presentar.
- Los desarrolladores manipulen y controlen con orden y seguridad sus enormes colecciones de archivos y componentes diferentes para cada aplicación.

Las organizaciones de desarrollo exitosas entienden que el control de cambios durante todo el ciclo es la clave para asignar prioridades a las actividades del equipo, así como para controlar la complejidad inherente del desarrollo. Sin ello, el caos de los cambios tomará control de todo proyecto.

La Administración de la Configuración y Control de Cambios (SCM) es la disciplina de la ingeniería de software que comprende las técnicas y las herramientas que una compañía usa para administrar los cambios de los componentes de software. Cuando SCM se encuentra integrado a otras actividades del desarrollo (requerimientos, análisis y diseño, construcción, pruebas), Rational le denomina Unified Change Management (UCM)