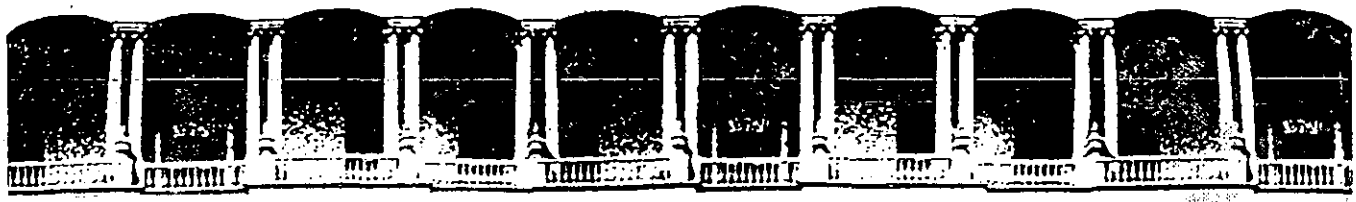


FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

TECNICAS PARA DESARROLLO DE SISTEMAS

LA PROGRAMACION ORIENTADA A OBJETOS

TEMA III

La programación orientada a objetos

La velocidad a la que avanza la tecnología de hardware de computadoras es sorprendente; año con año se logran avances que conducen a la construcción de computadoras más veloces, más compactas y más baratas. Sin embargo, en el aspecto de software no parece haber un desarrollo similar. Mientras los costos de hardware han disminuido continuamente, los de software han hecho lo contrario. La construcción de software no es una tarea fácil y en muchas ocasiones los proyectos de programación sobregiran los presupuestos de tiempo y dinero.

1.1 El problema del mantenimiento de software.

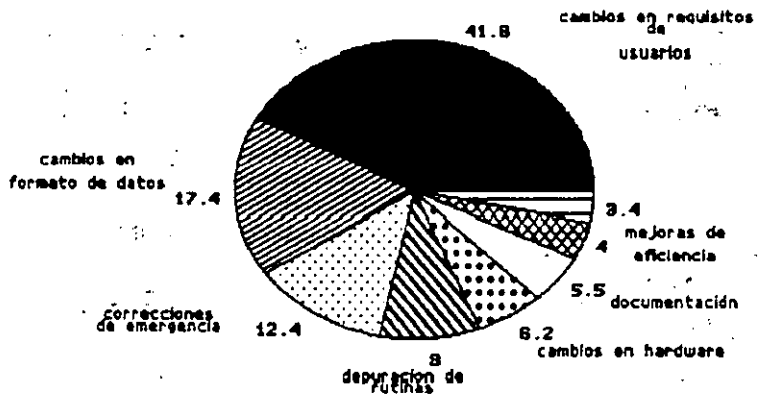
La construcción de software ha recibido la atención de los expertos desde hace mucho tiempo; en la década de los 70's se consiguieron avances significativos hacia el desarrollo de

metodologías para construir programas en forma sistemática y a bajo costo. Como resultado de esos esfuerzos surgieron técnicas que, como el diseño estructurado y el desarrollo descendente (top-down), durante mucho tiempo han sido las herramientas utilizadas por los programadores para construir software. Aunque dichas técnicas han sido empleadas durante mucho tiempo en proyectos realmente complejos, la mayoría de los ingenieros de software coinciden en afirmar que sufren de tres grandes deficiencias:

- los productos que resultan al emplear estas técnicas son poco flexibles.
- los programadores que las usan tienden a concentrarse en el diseño y la implementación inicial del sistema, sin tomar en cuenta su vida posterior.
- no alientan al programador a aprovechar el trabajo de proyectos anteriores.

La desventaja de utilizar una metodología que se concentra en el diseño inicial del sistema se hace evidente si se toma en cuenta que la vida útil de un producto de software puede ser cinco ó seis veces más grande que el lapso en que se desarrolla; por ejemplo, un sistema que se desarrolla en uno ó dos años puede mantenerse trabajando durante un período que va de cinco a quince años. Los gastos que se hacen durante este último período (*gastos de mantenimiento*) representan alrededor del 70% del costo total del

sistema. La figura 1 ilustra la forma en que se distribuyen estos gastos.



La gráfica muestra que cerca del 60% de los costos de mantenimiento de un sistema (alrededor del 42% del costo total) se tienen que hacer por cambios en las especificaciones del usuario ó en los formatos de los datos. Es por ello que la *extensibilidad* (la facilidad con que se modifica un sistema para que realice nuevas funciones) debe ser uno de los objetivos primarios de la etapa de diseño.

La fase de mantenimiento es tan importante que cualquier método de diseño debe tener como objetivo principal producir sistemas faciliten su propio mantenimiento. Pero, ¿cómo alcanzar este objetivo? Los expertos recomiendan una serie de actividades y heurísticas que pueden servir como guía durante el desarrollo del sistema. La tabla 1 agrupa tales actividades de acuerdo a la etapa

de desarrollo en que se deben realizar; más tarde nos ocuparemos con mayor cuidado de la *modularidad*, el *ocultamiento de información* y la *abstracción de datos*.

Actividades de análisis
Establecimiento de estándares (formatos de documentos, codificación, etc.).
Especificar procedimientos de control de calidad.
Identificar probables mejoras del producto.
Estimar recursos y costos de mantenimiento.
Actividades de diseño
Establecer la claridad y modularidad como criterios de diseño.
Diseñar para facilitar probables mejoras.
Usar notaciones estandarizadas para la documentación, algoritmos, etc.
Seguir los principios de ocultamiento de información, abstracción de datos y desconposición jerárquica de arriba hacia abajo.
Especificar efectos colaterales de cada módulo.
Actividades de implementación
Usar estructuras de una sola entrada y una sola salida.
Usar sangrado estandard en las diferentes estructuras.
Usar un estilo de codificación simple y claro.
Usar constantes simbólicas para asignar parámetros a las rutinas.
Proporcionar prólogos estandard de documentación en cada módulo.
Otras actividades
Desarrollar una guía de mantenimiento.
Desarrollar un juego de pruebas.
Proporcionar la documentación del juego de pruebas.

Tabla 1. Actividades que facilitan el mantenimiento de un sistema.

1.2 Reutilización de código.

La *reutilización de código* es otro de los factores que no se toma en cuenta en los métodos de diseño tradicionales. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente por él mismo

o por otros programadores.

Tradicionalmente se han empleado tres tipos de reutilización: de personal, de diseño y de código fuente. En la primera, a un proyecto nuevo se le asignan programadores que tienen experiencia en el desarrollo de proyectos semejantes; la segunda consiste en emplear el diseño de un sistema para desarrollar otro sistema similar; la última forma de reutilización se da cuando un programador utiliza parte un programa escrito con anterioridad para crear un nuevo programa. Sin embargo, estas tres formas de reutilización se han empleado en forma muy limitada, por lo que es necesario buscar técnicas más generales.

1.3 Modularidad.

La modularidad es una de las herramientas de diseño más poderosas para facilitar el desarrollo y mantenimiento de sistemas de software. La modularidad permite definir un sistema complejo en términos de unidades más pequeñas y manejables; cada una de esas unidades (ó *módulos*) se encarga de manejar un aspecto local de todo el sistema, interactuando con otros módulos para cumplir con el objetivo global.

La mayoría de los lenguajes de programación actuales alientan el uso de la modularidad: en lenguajes estructurados como C ó

Pascal, la modularidad se basa en el concepto de función (también llamada procedimiento o subrutina); en lenguajes más evolucionados, como Ada ó Modula-2, los módulos corresponden a conjuntos de funciones relacionados con las estructuras de datos que manipulan esas funciones (paquetes, en la terminología de Ada).

Sin embargo, para ser realmente útil, el concepto de módulo debe ser aún más sofisticado. Según Meyer, las propiedades de un módulo deberían ser las siguientes:

a) *Decomponibilidad*: Un método de diseño modular debe permitir descomponer un problema de diseño en subproblemas más pequeños que pueden resolverse en forma independiente.

b) *Componibilidad*: Una vez que se cuenta con un conjunto de módulos que realizan una función específica, se debe alentar al programador a usar esos módulos para construir nuevos programas.

c) *Comprensibilidad*: El lector de un programa o librería debe ser capaz de entender el funcionamiento de cada módulo sin necesidad de consultar el texto de otros módulos.

d) *Continuidad*: Un cambio pequeño en las especificaciones de un programa debe causar cambios en un sólo módulo ó en un conjunto pequeño de ellos.

e) *Protección*: Un error de ejecución en el funcionamiento de un módulo no debe expandirse hacia los demás módulos.

Estas cinco propiedades pueden alentarse con las siguientes estrategias:

a) Un módulo debe corresponder con una unidad sintáctica del lenguaje (una subrutina, un paquete, una *clase*); esta unidad debe poder ser compilada por separado, tal vez para ser almacenada en una librería (con esto se mejoran la componibilidad y comprensibilidad del sistema).

b) Los módulos deben tener pocas interfaces (medios de comunicación con otros módulos), y éstas deben ser pequeñas. Un número pequeño de interfaces aumenta la independencia de los módulos, lo cual hace más fácil el proceso de componer nuevos sistemas a partir de módulos prefabricados, ayuda a evitar que los errores en un módulo se propaguen por todo el sistema y hace que cada módulo de una sistema sea más comprensible.

c) Cada módulo debe ocultar su implementación y algoritmos internos al resto del sistema (*principio de ocultamiento de información*). No se debe permitir que un módulo modifique los elementos internos de otros módulos; sino que la comunicación entre ellos debe realizarse mediante interfaces explícitas y bien definidas (esto favorece la comprensibilidad y la protección modular).

1.4 La programación orientada a objetos.

La *programación orientada a objetos (OOP)* es un método de diseño que tiene como objetivo establecer técnicas de desarrollo de software para producir sistemas modulares. Un sistema orientado a objetos se puede entender fácilmente, por lo que su desarrollo, depuración y mantenimiento se facilitan en gran medida.

En primera instancia, la OOP se basa en una idea relativamente sencilla: un programa de computadora es un modelo que representa un subconjunto del mundo real; la estructura de ese programa se simplifica en gran medida si cada una de las entidades (u *objetos*) del problema que se está modelando corresponde directamente con un objeto que se pueda manipular internamente en el programa. Un ejemplo sencillo puede ser un programa de nómina: cualquier empleado de cierta empresa constituye una entidad real que tal vez se represente dentro de un programa con un conjunto de variables o con una estructura (o registro).

El proceso de representar entidades reales con elementos internos a un programa recibe el nombre de *abstracción de datos*. La abstracción de datos no se concentra en la representación interna de un objeto (un entero, una cadena de bits, un arreglo), sino en sus atributos (como el nombre, sueldo y edad de un empleado) y en las operaciones que se pueden realizar sobre ese

objeto (como calcular el sueldo de un empleado ó los impuestos que debe pagar). De esta forma, un *tipo de dato abstracto* se puede describir concentrándose en las operaciones que manipulan a los objetos de ese tipo, sin caer en los detalles de representación y manipulación de los datos.

La abstracción de datos es un concepto común en los lenguajes de programación modernos. Muchos lenguajes proporcionan un tipo de datos para números de punto flotante; cuando un programador utiliza datos de ese tipo, puede hacer operaciones como suma, multiplicación y exponenciación con esos datos, pero no es necesario que se preocupe por la representación de los números (p.e. cuatro palabras de máquina en las que se almacenan la mantisa y el exponente, junto con sus signos) ni por la forma en que el procesador realiza las operaciones (suma de enteros, comparación, desplazamiento lógico, etc.).

Sin embargo, en la mayoría de los programas, los objetos involucrados son mucho más complejos que un número de punto flotante. La programación orientada a objetos trata de describir tipos de datos de más alto nivel, por ejemplo listas, ventanas, menús, figuras geométricas, procesos industriales, etc.. Tomemos, por ejemplo, un tipo llamado STACK; el programador que defina variables de ese tipo (tal vez para evaluar una expresión

matemática ó para implementar el recorrido de un árbol) únicamente necesita tomar en cuenta la propiedad LIFO (último en entrar primero en salir) de los stacks, algunos atributos (p.e. *stack_lleno, tamaño*) y las operaciones que se pueden realizar sobre ellos (p.e. *push, pop, crea_stack*). Para ese programador no es necesario conocer la estructura de datos con la que se instrumenta el stack (p.e. un arreglo ó una lista ligada) ni los algoritmos con los que se implementan dichas operaciones.

La creación de tipos abstractos permite al programador adaptar el lenguaje de programación a sus necesidades específicas; el usuario de un lenguaje debe ser capaz de definir tipos de datos que se ajusten al problema que está resolviendo y que puedan ser manipulados como los tipos internos del lenguaje. Cuando el programador cuenta con esta facilidad, se puede concentrar en los objetos que manipula su sistema y las relaciones entre esos objetos, haciendo a un lado los detalles de representación y manipulación de los datos, para lograr una mejor comprensión del problema.

En la terminología del diseño orientado a objetos, un tipo abstracto es llamado *clase*. La OOP modela al mundo real utilizando objetos (instancias de una clase); el centro de atención son los datos. Este enfoque resulta particularmente exitoso porque durante

el ciclo de vida de un sistema, los datos que se manipulan sufren pocos cambios, mientras que las acciones que se deben realizar sobre esos datos cambian constantemente.

Una clase describe un conjunto de objetos semejantes. Dicha descripción se hace en dos partes: los datos que especifican las propiedades de los objetos de esa clase (llamados *atributos*) y las funciones que manipulan esos datos (llamadas *métodos*). Un objeto puede recibir mensajes de otros objetos; entonces, debe escoger uno de sus métodos y dar una respuesta basándose en los datos que representan su estado (los atributos pueden ser modificados por los métodos).

Cada objeto cuenta con *elementos privados*, que sólo pueden ser usados por objetos de su misma clase, y *elementos públicos*, a los que tiene acceso cualquier otra entidad. Los elementos públicos representan la *interface* de un objeto con su medio ambiente; únicamente esos elementos pueden modificar a los datos privados (los cuales representan el *estado* del objeto). Observe que este esquema se ajusta al principio de ocultamiento de información.

La programación orientada a objetos propone dos estrategias para la reutilización de código: la *composición* y la *herencia*. La composición permite definir una nueva clase de objetos mediante la

unión de un conjunto de clases ya existentes. Por ejemplo, una clase que permita definir objetos que simulen procesos industriales puede necesitar de una forma de medir el tiempo, que tal vez sea proporcionada por una clase llamada **cronómetro**.

Por otro lado, la herencia permite crear (derivar) una nueva clase basándose en otra clase más general. Una *clase derivada* adquiere todas las propiedades y métodos de la clase de la que se derivó (*clase base*). De esta forma, puede ser posible derivar una clase **pentágono** a partir de una clase **polígono**, de tal forma que la primera adquiera todos los atributos (color, centro, relleno, etc.) y métodos (dibujar, borrar, escalar, etc.) de la segunda, tal vez añadiendo nuevos elementos o modificando ligeramente los ya existentes. El ejemplo anterior se basa en la llamada *herencia sencilla*; la *herencia múltiple* proporciona un método para derivar una clase de un conjunto de ellas, por ejemplo un sistema de ventanas puede obtenerse de las clases *stack*, *ventana* y *editor*. Los procesos de composición y de herencia múltiple son muy parecidos, más adelante se tratará este aspecto.

Además de facilitar la reutilización de código, la herencia es el medio ideal para crear sistemas con una alta extensibilidad. Otra ventaja de esta técnica es que permite manipular objetos de clases diferentes como si fueran de la misma clase (*polimorfismo*),

con lo cual es posible definir interfaces uniformes para diferentes tipos de objetos.

1.5 Lenguajes orientados a objetos.

Las técnicas en las que se basa la programación orientada a objetos (ocultamiento de información, abstracción de datos, manejo automático de memoria, polimorfismo) eran conocidas y utilizadas por los ingenieros de software desde hace muchos años; lenguajes como Ada ó Modula-2 alientan a los programadores a usar algunas de esas técnicas.

Sin embargo, son pocos los lenguajes que brindan todas las facilidades para escribir programas orientados a objetos. Existen diversas opiniones acerca de cuáles deben ser dichas facilidades; para Bertrand Meyer (autor de Eiffel, uno de los lenguajes orientados a objetos más populares) deben ser las siguientes:

a) Estructura modular basada en objetos. Los sistemas se deben modularizar tomando como base sus estructuras de datos.
b) Abstracción de datos. El lenguaje debe permitir al programador definir tipos de datos abstractos.
c) Manejo de memoria automático. La memoria ocupada por objetos cuya utilidad ha terminado debe ser liberada por mecanismos internos al lenguaje, sin intervención del programador.

d) Clases. Cada tipo no simple es un módulo y cada módulo es un tipo.

e) Herencia. El lenguaje debe permitir definir clases como extensiones o restricciones de otras clases.

f) Polimorfismo y asociación dinámica de tipos. Las entidades internas de un programa deben poder manejar conjuntos de objetos de diferentes clases de la misma forma en que manejan conjuntos de objetos iguales. Una operación puede comportarse de varias formas de acuerdo a la clase de objeto que manipula.

g) Herencia múltiple. Una clase debe poder ser derivada de más de una clase.

Entre los lenguajes orientados a objetos más populares, se encuentran Simula67, un lenguaje de simulación con facilidades para manipular eventos discretos; Smalltalk, que se ha usado principalmente para desarrollar interfaces de usuario gráficas y Eiffel, que se ha aplicado en áreas diversas. Los lenguajes orientados a objetos no habían tenido hasta recientemente una aceptación amplia entre la comunidad de programadores. Características como el manejo de memoria automático y la asociación dinámica de tipos imponen sobrecargas demasiado grandes a la ejecución de programas escritos en dichos lenguajes. Recientemente han surgido dos estrategias para disminuir esa sobrecarga: una de ellas consiste en utilizar lenguajes orientados

a objetos para desarrollar los componentes de más alto nivel de un sistema y lenguajes funcionales para escribir las partes de bajo nivel críticas para la ejecución (una forma común de este tipo de combinaciones es utilizar Smalltalk y C). La otra estrategia consiste en desarrollar nuevos lenguajes que no proporcionen todas las facilidades de la programación orientada a objetos, pero que no impongan sobrecargas de ejecución demasiado altas (a estos lenguajes se les ha llamado *híbridos*); resultados de este enfoque son lenguajes como C Objetivo, C++ y algunas versiones de Pascal (como el Turbo Pascal 5.5 de Borland).

1.6 El lenguaje de programación C++.

C++ es una extensión orientada a objetos del lenguaje C. El objetivo principal de C++ es disminuir u ocultar la complejidad de cualquier proyecto de programación de tal forma que un sólo programador, o un grupo pequeño de ellos, pueda desarrollarlo y darle mantenimiento con poca dificultad.

El diseño inicial de C++ se realizó a inicios de los 80's en los laboratorios Bell de la AT&T, dirigido principalmente por Bjarne Stroustrup. Stroustrup habla acerca de los origenes del lenguaje: "El nombre C++ es una invención reciente (verano de 1983). Desde 1980 se venían usando versiones previas del lenguaje, llamadas colectivamente 'C con clases'. El lenguaje se inventó

originalmente porque el autor deseaba escribir ciertas simulaciones manejadas por eventos, para lo cual Simula67 hubiera sido ideal, excepto por consideraciones de eficiencia. 'C con clases' se utilizó en proyectos de simulación más grandes en los que se debían usar tiempo y espacio mínimos... C++ se instaló por primera vez fuera del grupo de investigación del autor en Julio de 1983... El nombre simboliza la naturaleza evolutiva del lenguaje, '++' es el operador de incremento de C. El nombre C+ es un error de sintaxis, porque ya ha sido usado para otro lenguaje. El lenguaje no se llamó D porque es una extensión de C y no trata de remediar sus problemas quitando características".

C fue elegido como el lenguaje base de C++ por varias razones: C es un lenguaje de propósito general conocido por una gran cantidad de programadores; existen compiladores de C para un vasto conjunto de computadoras; C es un lenguaje expresivo y eficiente y, finalmente , se ha escrito una gran cantidad de código y librerías en C que no pueden ser desperdiciadas. La popularidad de C facilita el aprendizaje de C++, pues no es necesario aprender un nuevo lenguaje desde cero, sino que un programador puede ir aprendiendo nuevas características del lenguaje conforme las necesite. Como los dos lenguajes son compatibles, es posible utilizar todo el código escrito en C desde C++; la compatibilidad tiene otra ventaja: se puede escribir un compilador de C++ en poco

tiempo aprovechando el 'back end' de un compilador de C.

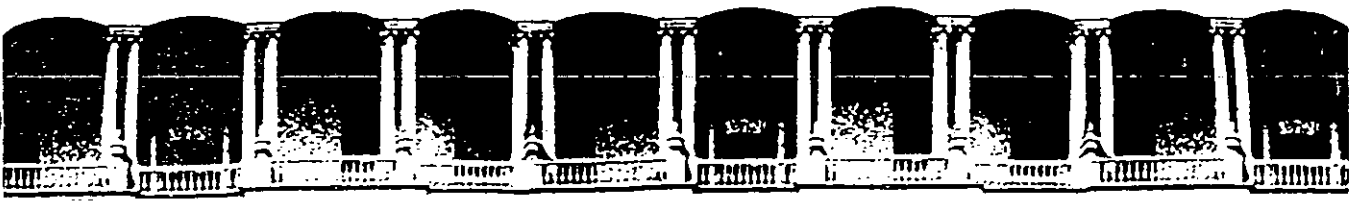
El diseño de C++ tuvo desde sus inicios el compromiso de conservar en la medida de lo posible la eficiencia a tiempo de corrida de C. Aunque la sobrecarga de un programa en C++ es mayor que la de su equivalente en C, todos los elementos del primero han sido cuidadosamente diseñados para minimizar esa diferencia (en programas grandes, el código en C++ tiende a ser más pequeño, además de que la diferencia de rendimientos es despreciable).

C++ es un lenguaje que ha evolucionado y sigue evolucionando rápidamente; los cambios en el lenguaje han surgido principalmente de problemas encontrados por sus usuarios o de innovaciones propuestas por el autor. La liberación 1.0 del lenguaje fue la especificada por Stroustrup en "El lenguaje de programación C++" (1986); más tarde se publicaron las liberaciones 1.1 y 1.2. La popularización del lenguaje se inició con la liberación 1.2, sin embargo, al crecer la población de programadores de C++ se hizo evidente la necesidad de nuevas mejoras. La siguiente versión de C++ (2.0) apareció en el verano de 1989 y es la que se encuentra vigente. Actualmente no existe una definición formal de C++, aunque se aceptan como estándares las versiones de la AT&T; sin embargo, se espera que para fines de 1992, la ANSI publique un estandard del lenguaje.

C++ se ha empleado en gran medida en ambiente Unix; sin embargo, también existen compiladores que corren bajo otros sistemas operativos. En el ambiente MSDOS, los compiladores más populares son los por Zortech, Glonckpesfield y Borland.

El lenguaje C++ se utiliza actualmente en el desarrollo de manejadores de bases de datos, sistemas operativos, compiladores, sistemas de comunicación, productos de CASE, redes y robótica.

Los programas de este texto fueron probados utilizando el TurboC++ de Borland; aunque en ellos sólo se utilizaron características estándares, el lector debe estar consciente de que tal vez se necesiten pequeños cambios para correrlos utilizando otro compilador. Lo mismo se aplica para la descripción del lenguaje que se hace aquí.



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

TECNICAS PARA DESARROLLO DE SISTEMAS

EL MODELO DE ENTIDADES Y ASOCIACIONES

El Modelo de Entidades y Asociaciones

El Modelo de Entidades y Asociaciones es un proceso descendente usado para simplificar el procedimiento de diseño de Bases de Datos en circunstancias reales, en donde puede haber un gran número de atributos por considerar y más de una relación entre los atributos.

A.1 Conceptos básicos

El Modelo de Entidades y Asociaciones está basado en la percepción de un mundo real que se compone de un conjunto de objetos básicos llamados **Entidades**, y de **Asociaciones** entre esos objetos. El siguiente párrafo intenta clarificar esta idea.

Cuando una Empresa maneja PRODUCTOS que son ENTREGADOS por un DISTRIBUIDOR, cada uno de esos productores o distribuidores es una Entidad, y la entrega de un PRODUCTO por un DISTRIBUIDOR representa una Asociación. El conjunto de todos los productos que maneja la empresa es entonces un "conjunto de Entidades", y todas las Asociaciones forman un "conjunto de Asociaciones". Sin embargo, por simplicidad, se denomina Entidad a la representación del conjunto de Entidades y Asociación a la representación del conjunto de Asociaciones.

El procedimiento de modelado es el siguiente:

1. Seleccionar las Entidades (como cliente, parte, o proveedor) y las Asociaciones entre ellas (como orden de partes por clientes, partes proporcionadas por el proveedor, etc.) que están dentro del Alcance de Integración de la Empresa.

2. Asignar atributos a esas entidades y Asociaciones, de tal forma que se puedan obtener tablas bien normalizadas.

Hasta aqui han entrado en juego principalmente tres conceptos: Entidad, Asociación y Atributo. A continuación se intentan definir.

Entidad. Es un objeto que existe y que es distinguible de otros objetos. Es algo (persona, lugar, objeto, concepto) a lo que la Empresa le reconoce poder existir en forma independiente y que puede ser definido en forma única. Por ejemplo: cliente, máquina, contrato.

Atributo. Es una propiedad de una Entidad. Número, nombre y RFC pueden ser atributos de la Entidad cliente.

Asociación. Es el vinculo que existe entre dos o más Entidades. Por ejemplo, la Entidad departamento puede asociarse con la Entidad empleado vía la Asociación emplea. Normalmente una Empresa utiliza sujetos para describir Entidades y verbos para describir Asociaciones.

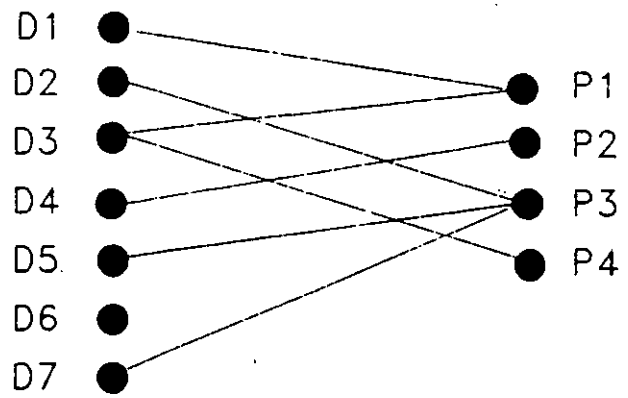


Figura A.1 Diagrama de ocurrencias DISTRIBUIDOR-VENDE-PRODUCTO.

Por definición, toda ocurrencia de una Entidad debe ser identificable en forma única. Es por lo anterior que se debe encontrar un identificador de la Entidad o llave. Existen varios tipos de llaves:

Súper llave. Es un conjunto de uno o más atributos que, tomados en conjunto, permiten identificar en forma única una Entidad dentro del conjunto de Entidades.

Llave candidato. El concepto de Súper llave no es suficiente, puesto que puede contener atributos extraños. Las Súper llaves con el menor conjunto de atributos se conocen como llaves candidato y es posible que existan diferentes conjuntos de atributos que puedan servir como llaves candidato.

Llave primaria. Una llave primaria es una llave candidato que ha sido seleccionada por el diseñador de la base de datos como

el medio de identificar Entidades. Durante el proceso de modelado se puede hablar, más bien, de llaves candidato, puesto que es posible que en algún momento del proceso se decida seleccionar otra llave como primaria.

A.2 Diagramas de Entidades y Asociaciones

El modelado de Entidades y Asociaciones cuenta con una herramienta gráfica para cumplir sus objetivos. El proceso se realiza dibujando diagramas conocidos como Diagramas de Entidades y Asociaciones (DEA). Las convenciones al dibujar DEA son:

1. Las Entidades serán representadas por rectángulos.
2. Las Asociaciones serán rombos.
3. Las líneas de conexión mostrarán qué Entidades son vinculadas por cuál Asociación.
4. Los atributos de las Entidades y las Asociaciones se muestran como círculos o elipses conectados al rombo o rectángulo correspondiente.
5. El grado de la Asociación será representado por 1, M ó N, sobre las líneas de conexión.
6. El grado de pertenencia se indicará terminando la correspondiente línea de conexión dentro de un pequeño rectángulo que forme parte de la Entidad.

Tanto los grados de asociación como las clases de pertenencia serán de utilidad al llevar el modelo de datos a un esquema conceptual. La información que proporciona define cómo estarán organizados los datos en la Base de Datos.

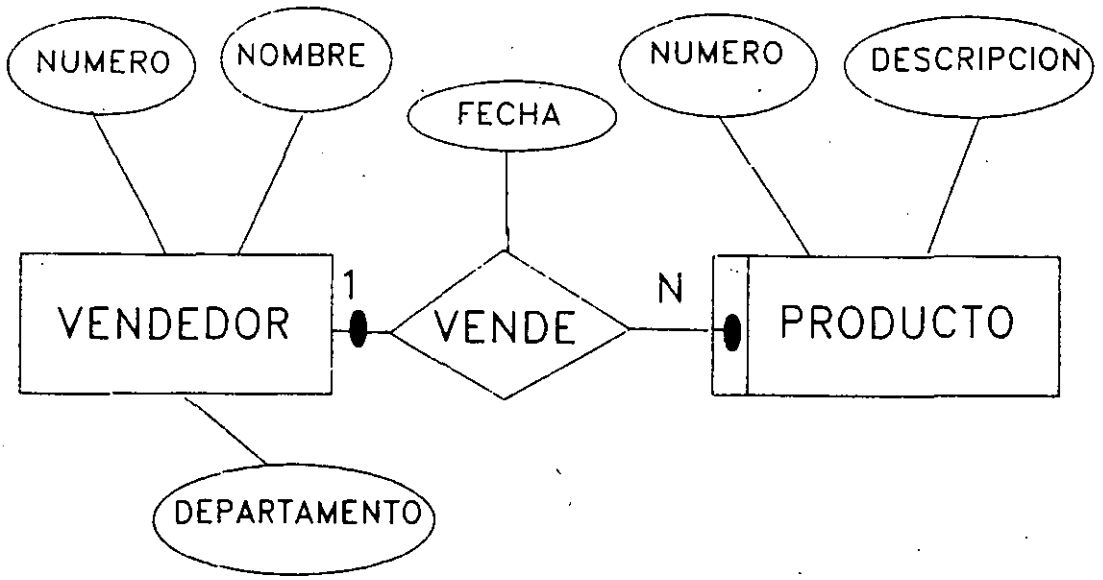


Figura A.2 Ejemplo de un DEA.

Grado de una asociación. Representa, en forma cualitativa, el número de ocurrencias de una entidad con las que puede estar asociada una ocurrencia de otra entidad. Una asociación puede tener tres tipos de grados:

TIPO		PUEDE INCLUIR
1:1	(uno a uno)	1:0 0:1 1:1
1:N	(uno a muchos)	1:0 0:1 1:1 1:N
M:N	(muchos a muchos)	1:0 0:1 1:1 1:N N:1 M:N

En un DEA, los grados de asociación se indican junto al rectángulo que representa cada Entidad, escribiendo un 1 una N o una M, según sea el caso correspondiente.

El grado de asociación puede ser obtenido de las reglas de empresa. Las reglas de empresa son definiciones que se obtienen del análisis de datos. Por ejemplo:

"Un conferencista puede dictar muchos cursos"
"Un curso sólo puede ser dictado por un conferencista"

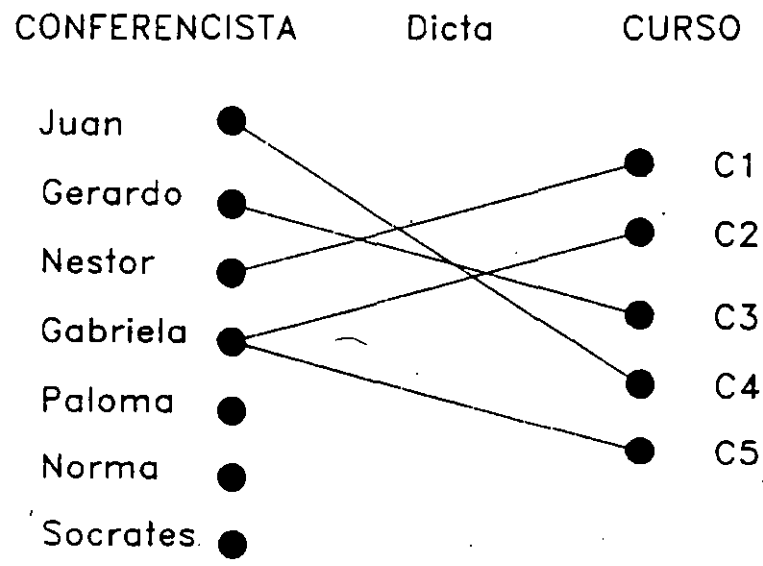


Figura A.3 Un conferencista puede dictar muchos cursos. Un curso sólo puede ser dictado por un conferencista.

Los grados de asociación definen la capacidad de determinación de una entidad hacia otra, de tal forma que en una asociación 1:1 existe determinación en ambos sentidos, en una asociación 1:N sólo existe determinación de la entidad de grado N hacia la de grado 1, y en asociaciones M:N no existe determinación.

Los grados que incluye una categoría de asociación pueden ser reducidos con reglas de empresa más estrictas, como:

"Un conferencista debe ofrecer varios cursos"
"Un curso debe ser ofrecido por un conferencista"

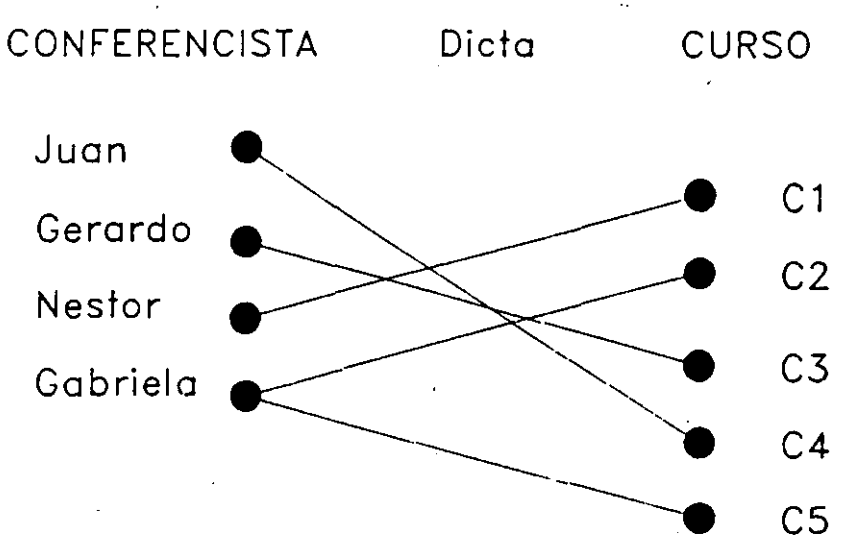


Figura A.4 Un conferencista debe ofrecer varios cursos. Un curso debe ser ofrecido por un conferencista.

Clases de pertenencia. Las reglas de empresa que excluyen Asociaciones 1:0 se traducen como la obligatoriedad de que una entidad participe en una Asociación con otra Entidad. Una Entidad con obligatoriedad se puede considerar como una Entidad débil puesto que para existir depende de la existencia de otra Entidad.

La clase de pertenencia se indica con un punto en la línea de conexión, que está dentro del rectángulo que representa la Entidad, cuando ésta tiene obligatoriedad; o fuera de él, cuando ésta no la tiene.

Asociaciones multitudinarias. Conocidas como asociaciones N-arias, son asociaciones que en un momento existen entre más de dos Entidades. El Modelo de Entidades y Asociaciones representa estos casos relacionando las Entidades involucradas con un mismo rombo, que representa la Asociación.

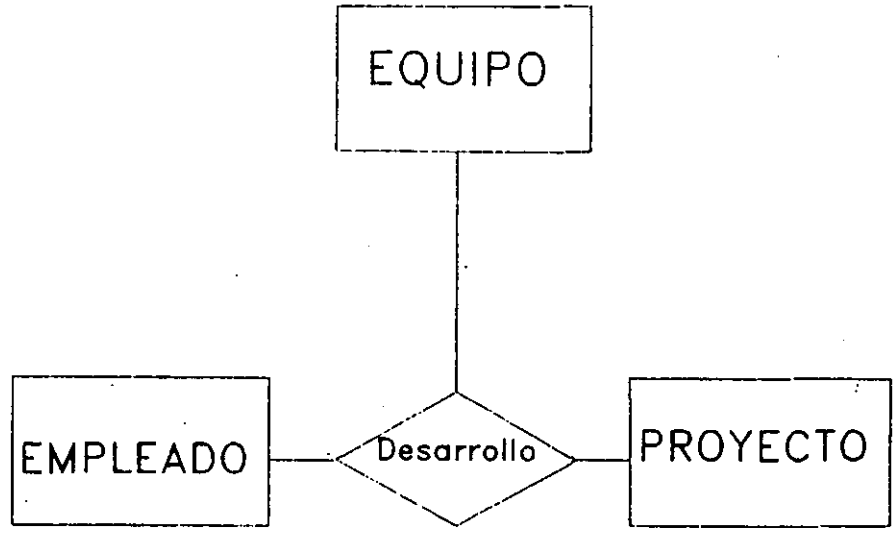


Figura A.5 Asociación ternaria PARTE-PROVEDOR-ALMACEN.

En las asociaciones multitudinarias no es de mucha utilidad especificar el grado de asociación y las clases de pertenencia; al final, la forma de organizar los datos es la misma. Sin embargo, algunas veces se incluye esa información como parte de la documentación gráfica que el DEA representa para el sistema.

Cuando la asociación se da entre una gran cantidad de Entidades, para evitar la pérdida de normalización algunos autores sugieren trasformar la Asociación en una nueva Entidad agregada, creando nuevas asociaciones entre la Asociación trasformada y las Entidades.

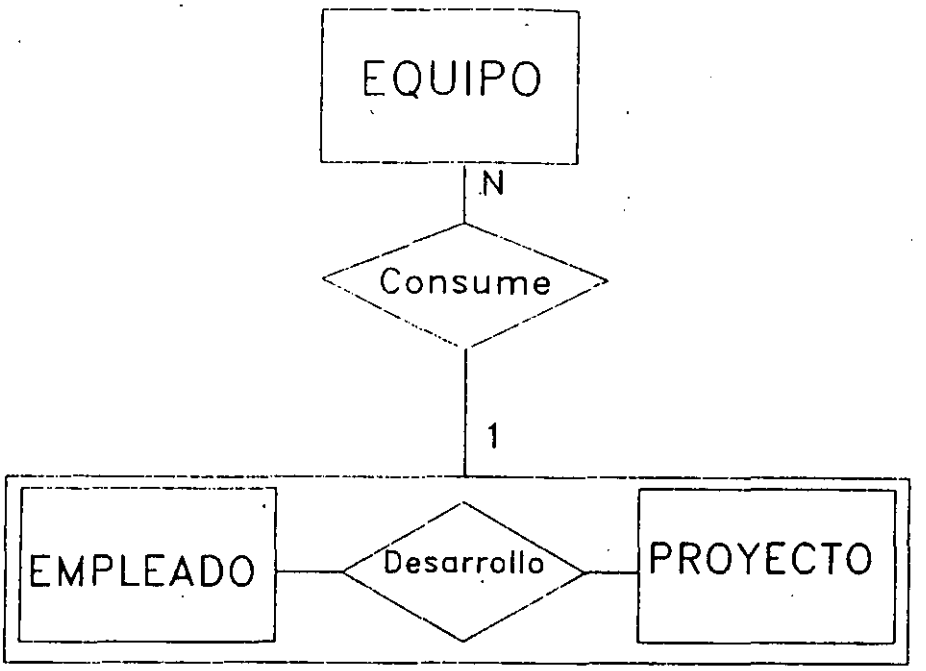


Figura A.6 Asociación multitudinaria trasformada.

Asociaciones recursivas. Son las que se dan entre Entidades del mismo conjunto de Entidades. Un ejemplo lo podemos ver considerando al conjunto de Entidades CIUDADANO y la Asociación MATRIMONIO, que debe ser entre dos CIUDADANOs. En estos casos el modelo se realiza exactamente igual a los casos no-recursivos, pero se debe indicar el papel que juega cada Entidad en la Asociación.

Generalización, especialización y agregación. Son herramientas de diseño utilizadas para facilitar la representación de la realidad y facilitar su manipulación. Por ejemplo, algunas veces será apropiado hablar de una PERSONA sin mencionar sus características, mientras que en otras se requerirá verla con más detalle, incluyendo su nombre o fecha de nacimiento.

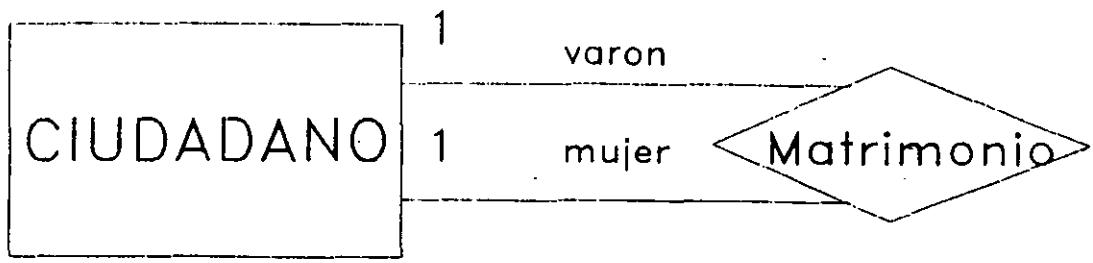


Figura A.7 Asociación recursiva CIUDADANO, MATRIMONIO.

La agregación da al diseñador la habilidad de descomponer objetos gradualmente en sus componentes detalladas, o de agregarlos en objetos de más alto nivel. Es un hecho que cuando se estudian los resultados que arroja el análisis de datos de la Empresa, se agregan datos aislados para obtener Entidades.

La generalización tiene que ver con la asignación de roles. Permite que los objetos sean separados en varios objetos especializados. Por ejemplo, un CAUSANTE puede ser especializado en CAUSANTE MAYOR y CAUSANTE MENOR para resaltar las diferencias que existen entre ambos tipos de causante. Inversamente, MARIDO y ESPOSA pueden ser generalizados como un CIUDADANO para marcar la igualdad de atributos que ambos tienen.

En un DEA la generalización y la especialización se indican con un triángulo que inscribe a la frase "ES UN", pudiéndose leer, por ejemplo, CAUSANTE ES UN CAUSANTE MENOR y un CAUSANTE MAYOR. La diferencia entre una y otra abstracción se indica con el grosor de las líneas que unen al triángulo con las Entidades

involucradas, líneas gruesas representarán generalización, mientras que las delgadas indican especialización.

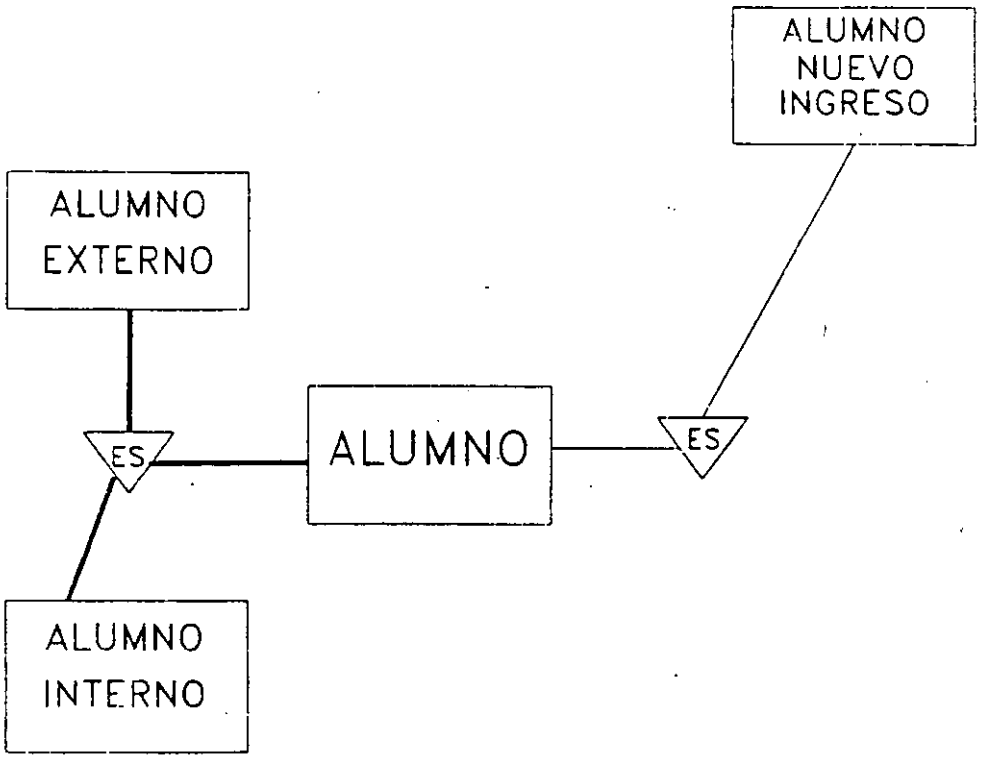


Figura A.8 Ejemplos de generalización y especialización. La línea gruesa indica generalización.

A.3 Mapeo de DEA a esqueletos y tablas

Como se anotó en la sección anterior, los atributos de Entidades y Asociaciones se muestran como elipses conectadas a los rombos o rectángulos. Sin embargo, en un DEA en donde existan muchas Entidades y Asociaciones, el número de círculos y elipses puede ser tan aparatoso que lo haría difícil de dibujar e incluso de comprender.

Para evitar el uso de elipses es generalmente conveniente representar a los atributos en tablas que se anexan al diagrama, adelantando incluso el camino hacia la implantación dentro de un ambiente Relacional.

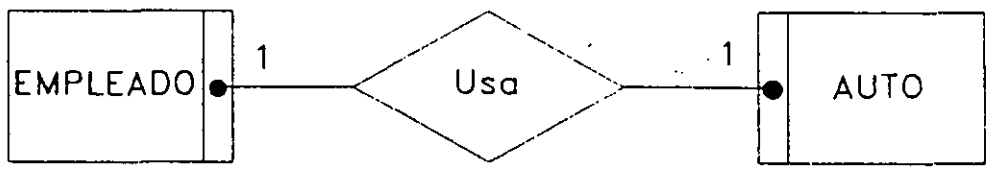
La primera etapa de mapeo es la obtención de esqueletos de las tablas que componen el sistema. El esqueleto de una tabla se compone de: el nombre de la tabla, que usualmente es el nombre de la Entidad o Asociación; una lista de atributos mínimos que debe contener esa tabla, que por lo regular son una llave candidato y las llaves foráneas necesarias para mantener el vínculo con otras tablas; y grupos de tres puntos, que indican la futura presencia de otros atributos de la tabla. La llave candidato se coloca al principio de la lista y se subraya para indicar su calidad de identificador de la Entidad.

La segunda etapa es la asignación del resto de atributos, colocándolos en la tabla que les corresponde, y cumpliendo siempre con las reglas de normalización (§B). Es un hecho que el conjunto de tablas resultantes puede ser implantado directamente dentro del ambiente de un manejador de bases de datos relacionales, puesto que cada tabla será una relación bien normalizada.

Obtención de esqueletos. Atendiendo a cada Asociación, y observando los correspondientes grados de asociación y clases de pertenencia, se pueden obtener directamente los esqueletos de las tablas que corresponden al diagrama. A continuación se muestra esto último con algunos ejemplos.

Representación de Asociaciones 1:1

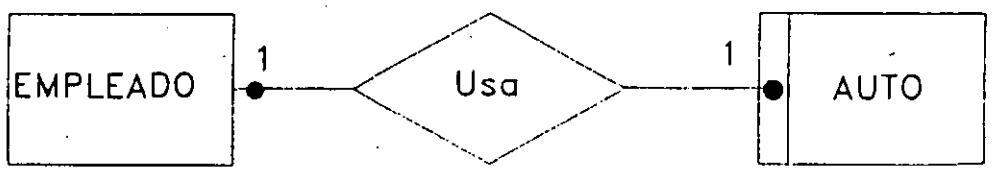
Obligatoriedad para ambas Entidades:



EMPLEADO(#empleado, ... ,#auto, ...)

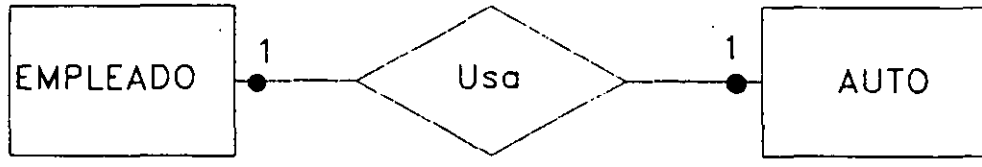
#empleado y #auto son los identificadores de cada Entidad.

Obligatoriedad sólo para una Entidad:



EMPLEADO(#empleado, ...)
AUTO(#auto, ... , #empleado)

No obligatoriedad para las dos Entidades:



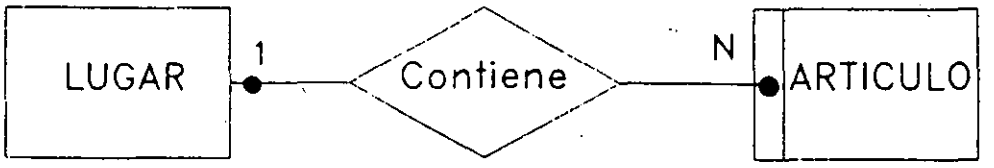
Se crea una tabla para la Asociación.

EMPLEADO(#empleado, ...)
AUTO(#auto, ...)
Usa(#empleado,#auto, ...)

En este caso, la llave candidato de la tabla de la Asociación puede ser la llave candidato de cualquiera de las Entidades.

Representación de Asociaciones 1:N

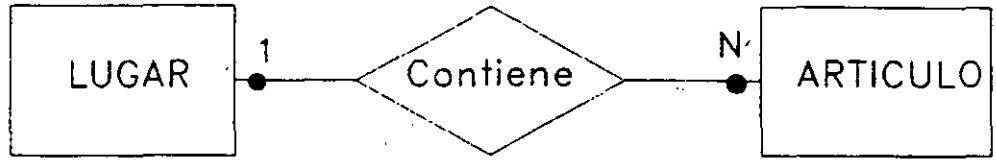
Obligatoriedad en la Entidad de grado N:



LUGAR(nombre, ...)
ARTICULO(#artículo, ..., nombre)

Nótese que la llave candidato de la Entidad de grado 1 se convierte en llave foránea de la tabla que representa la Entidad de grado N, en ese orden. De otra forma la llave de ARTICULO podría tomar valores nulos.

Sin obligatoriedad en la Entidad de grado N:



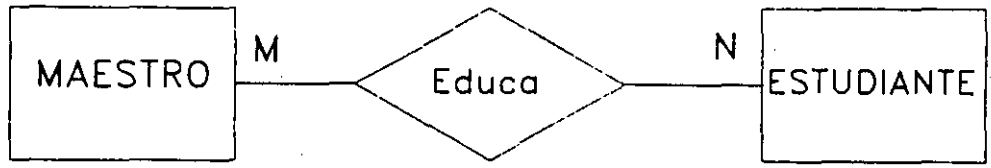
LUGAR(nombre, ...)
ARTICULO(#artículo, ...)
Contiene(#artículo, nombre, ...)

La llave candidato de la Entidad de grado 1 se convierte en la llave foránea de la Tabla que representa a la Asociación, en ese orden.

Los casos en que existe o no obligatoriedad en la Entidad de grado 1 no modifican nada.

Representación de Asociaciones M:N

No importa la existencia o no de obligatoriedad en las Entidades:

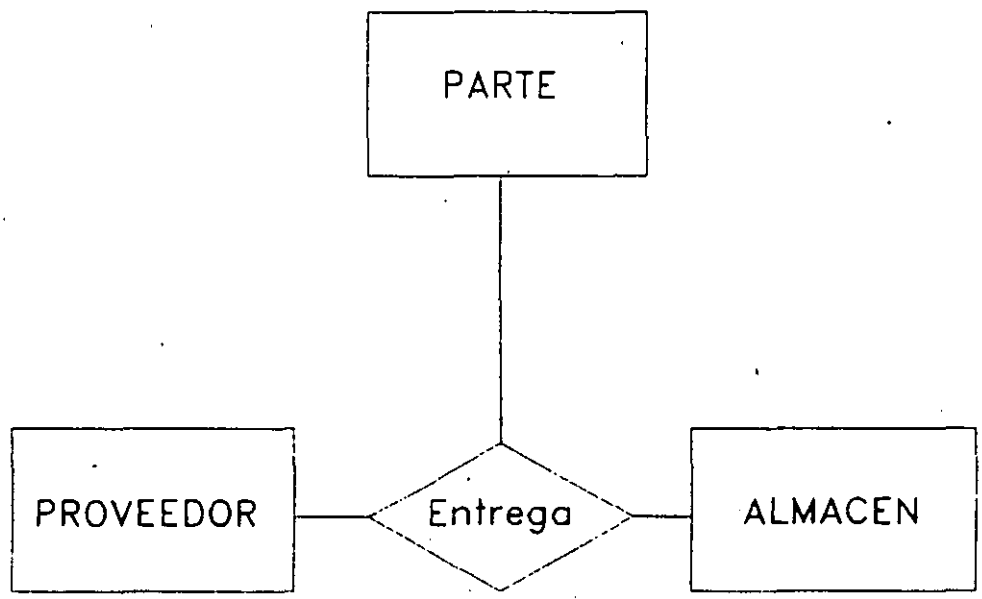


MAESTRO(nombre, ...)
ESTUDIANTE(#estudiante, ...)
Educa(nombre, #estudiante, ...)

La tabla que representa la Asociación debe tener como llave candidato la concatenación de las llaves de las Entidades.

Representación de Asociaciones Multitudinarias

No importan mucho los grados de asociación o de pertenencia, de cualquier forma se debe representar la Asociación y formar una llave primaria con la concatenación de los identificadores de las Entidades que participan.



PARTE(#parte, ...)
ALMACEN(#almacén, ...)
PROVEEDOR(#proveedor, ...)
Entrega(#proveedor,#parte,#almacén, ...)

Descomposición de Asociaciones M:N. Para algunos modelos o manejadores de bases de datos existe la imposibilidad de representar las Asociaciones M:N, debido por ejemplo a la incapacidad de concatenar llaves. En estos casos es posible descomponer una Asociación M:N en dos 1:N, lo cual no significa que dos Asociaciones 1:N deban representar una M:N. Ejemplo:

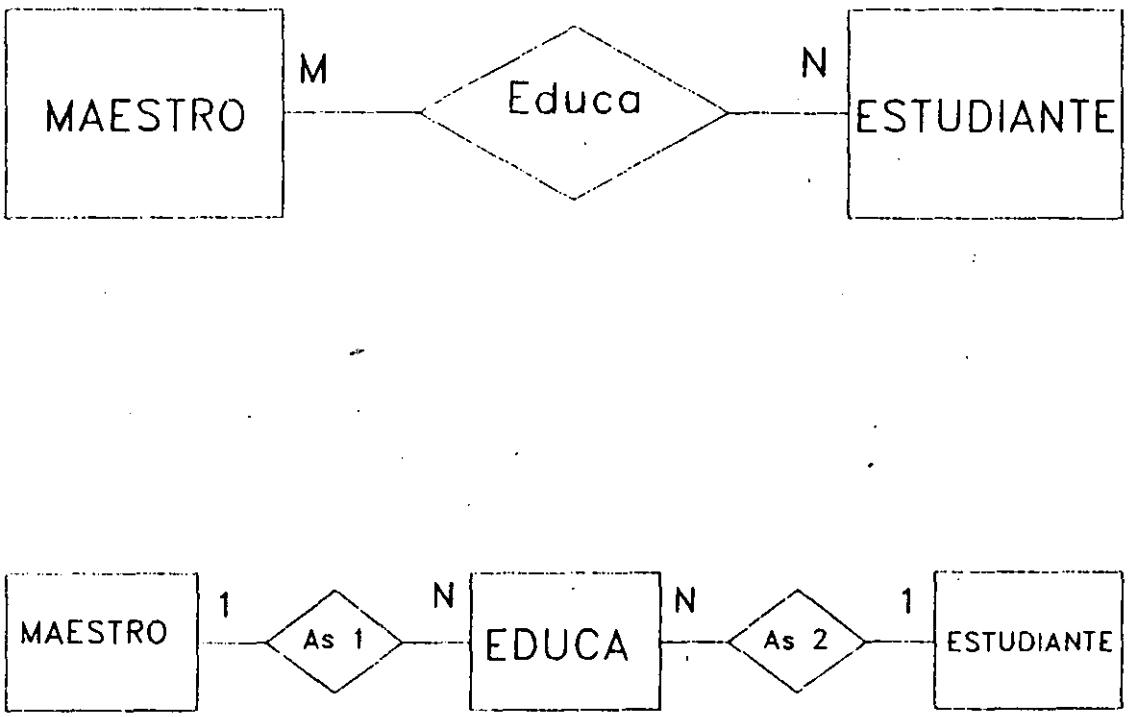


Figura A.16 Descomposición de una Asociación M:N en dos 1:N.

página intencionalmente blanca

Normalización

Normalización es una técnica desarrollada para asegurar que las estructuras de datos sean eficientes. Los beneficios de la Normalización son:

- Libera de dependencias indeseables de inserción, borrado y actualización.
- Minimiza la reestructuración de datos cuando se introduce algo nuevo. Se mejora la independencia de datos, permitiendo que las extensiones a la base de datos tengan poco o ningún efecto sobre los programas o aplicaciones que tienen acceso a ella.
- No se introducen restricciones artificiales a las estructuras de datos.

Aunque se han definido más estados de normalización, sólo se han aceptado ampliamente tres. Estos se conocen como primera, segunda, y tercera formas normales, o 1NF, 2NF y 3NF respectivamente.

El normalizar es un proceso ascendente, en el que se parte de un universo de relaciones y atributos, y se avanza de forma en forma hasta llegar a la tercera forma normal.

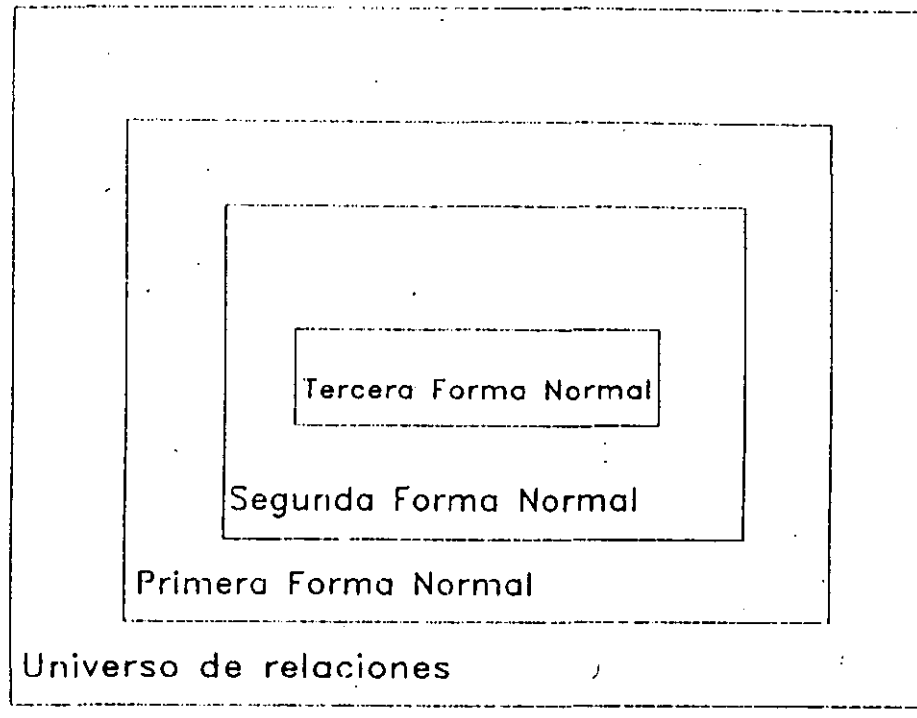


Figura B.1 El universo de las relaciones.

Las etapas de normalización se muestran adelante con un ejemplo, dada la relación no normalizada:

ORDEN (#orden, fecha, #proveedor, nombre_proveedor, dirección_proveedor, #producto, descripción_producto, cantidad_producto, precio_total_producto, precio_total_orden)

Primera forma normal. Un registro en primera forma normal no incluye grupos repetidos.

En la relación ORDEN, se observa que para una misma orden habrá varios productos, por lo que #producto y otros atributos serán grupos repetidos. En 1NF habría que separar:

ORDEN (#orden, fecha, #proveedor, nombre_proveedor, dirección_proveedor, precio_total_orden)

PRODUCTO_ORDENADO (#orden, #producto, descripción_producto, precio_producto, cantidad_producto, precio_total_producto)

Segunda forma normal. Cada atributo depende de la totalidad de la llave, y no de sólo una parte de ella.

Se puede observar en PRODUCTO_ORDENADO que descripción_producto depende sólo de #producto, y no tiene que ver con #orden. En 2NF quedaría:

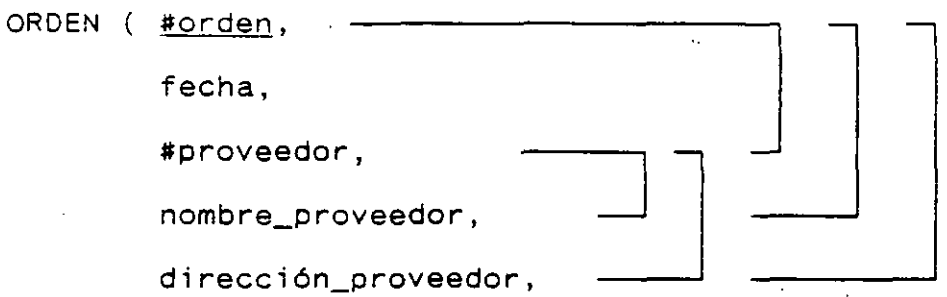
ORDEN (#orden, fecha, #proveedor, nombre_proveedor, dirección_proveedor, precio_total_orden)

PRODUCTO (#producto, descripción_producto, precio_producto)

PRODUCTO_ORDENADO (#orden, #producto, cantidad_producto, precio_total_producto)

Tercera forma normal. Cada atributo no primo (que no es parte de la llave) no tiene dependencia transitoria sobre la llave.

En la relación ORDEN se presenta este problema:



De modo que las tablas en 3NF quedarían como:

ORDEN (#orden, fecha, #proveedor, precio_total_orden)

PROVEEDOR (
#proveedor, nombre_proveedor, dirección_proveedor
)

PRODUCTO (
#producto, descripción_producto, precio_producto
)

PRODUCTO_ORDENADO (
#orden, #producto, cantidad_producto,
precio_total_producto
)

SQL

SQL fue utilizado primeramente en el manejador de bases de datos *System R* y a la fecha, se ha convertido en el lenguaje estándar para la manipulación de bases de datos. El nombre SQL proviene de las siglas de *Structured Query Languaje* (Lenguaje de consulta estructurado) y comunmente se pronucia "sicuel".

La estructura básica de una expresión de SQL consiste de tres cláusulas: **select**, **from** y **where**.

- La cláusula **select** corresponde a la operación de proyección del álgebra relacional. Se emplea para enlistar especificamente los atributos deseados, como resultado de una consulta.
- En la cláusula **from** se indica una lista de tablas sobre las que se ejecutará la expresión.
- La cláusula **where** corresponde con el predicado de selección del álgebra relacional. Consiste de un predicado que involucra atributos que aparecen en las tablas especificadas en la cláusula **from**.

Se debe tener cuidado de no confundir la interpretación que se da de la palabra "select" en SQL, de la que se le da en el álgebra relacional. Estrictamente nablando, en la cláusula

select se hace una proyección, mientras que la selección se realiza de acuerdo con la cláusula "where".

Una consulta típica de SQL se ve como:

```
select   A1, A2, ..., An
from     r1, r2, ..., rm
where    P
```

Los A_is representan atributos, los r_is son relaciones y P es un predicado. Esta consulta es equivalente a la expresión del álgebra relacional:

$$\pi_{A1,A2,\ldots An}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Cuando es omitida la cláusula **where**, el predicado P se considera verdadero. La lista de atributos $A_1, A_2, \ldots, A_n$ puede ser remplazada por un asterisco (*) para seleccionar a todos los atributos de las relaciones o tablas que aparecen en la cláusula **from**.

SQL forma el producto cartesiano de las relaciones nombradas en la cláusula **from**, realiza una selección utilizando el predicado de la cláusula **where** y proyecta los atributos de la cláusula **select** en el resultado. En la práctica, SQL puede convertir la expresión en una forma equivalente que se pueda procesar en forma más eficiente.

Al final, el resultado de una consulta SQL será siempre una relación o tabla. Por ejemplo, como resultado de la consulta

```
select    cuenta
from      alumno
```

se tendrá una tabla de un atributo, con tantos número de cuenta como tuplos tenga la tabla alumno.

Si se deseara eliminar las ocurrencias duplicadas se emplearía una consulta como la siguiente:

```
select distinct   cuenta
from  alumno
```

SQL permite realizar las operaciones de unión, intersección y diferencia equivalentes a los operadores U, $\cap$ y - del álgebra relacional, con **union**, **intersect** y **minus**.

Por ejemplo, para tener a los alumnos que tienen promedio de 10.0 entre los alumnos internos y alumnos externos, se haría una consulta como

```
(select nombre
 from   alumno_interno
 where  promedio = 10.)
union
(select nombre
 from   alumno_externo
 where  promedio = 10.)
```

La operación *join* se realiza utilizando el predicado de la cláusula **where**, y tomando provecho de que el producto cartesiano se efectúa siempre que se involucran más de una tabla en la operación. Ejemplo:

```
select    departamento.nombre, empleado.nombre
from      departamento, empleado
where     departamento.numero = empleado.no_depto
```

producirá una tabla con atributos "nombre de departamento" y "nombre de empleado".

Algo más complicado sería solicitar los nombres de empleados, acompañados por el nombre de su departamento, de aquellos empleados que actualmente tengan 15 años de antigüedad. La consulta sería

```
select    departamento.nombre, empleado.nombre
from      departamento, empleado
where     departamento.numero = empleado.no_depto
          and
          empleado.antig = 15
```

en donde se introduce uno de los operadores lógicos and, or, y not. De esa forma, el predicado es: "que el número de departamento del empleado sea igual al número del departamento y que la antigüedad del empleado sea igual a quince".

Además, tomado del cálculo relacional, SQL facilita la prueba de membría (que sea miembro de un conjunto) de alguna ocurrencia de un atributo. Considere la consulta: "Presente el nombre y número de cuenta de los alumnos con promedio igual a 9 y que pertenecen al grupo número 23", en donde se puede emplear la conectiva in para verificar si un alumno con el promedio descrito se encuentra en tal grupo. Una consulta podría verse como:

```
select    cuenta, nombre
from      alumno
where     promedio = 9.0
          and
          cuenta in   (
              select  cuenta
              from    inscrito
              where   grupo = 23)
```

Es además posible probar la membría en una relación arbitraria. SQL utiliza la notación $\langle v_1, v_2, \ldots v_n \rangle$ para denotar un tuplo de orden n que contiene los valores v_1, v_2, ... v_n. Empleando esta notación se puede escribir una consulta como la siguiente:

```
select    clave_cliente
from      prestamo
where     clave_sucursal = 9
          and
          < clave_sucursal, clave_cliente > in
          ( select    clave_sucursal, clave_cliente
            from      deposito )
```

De igual forma es posible emplear el conector not in, tal como se intuye.

Cuando se requiere comparar dos tuplos en la misma relación, se hace necesario el concepto de *variable tuplo*. Una variable tuplo debe ser asociada con una relación en particular y definida en la cláusula from.

Para ejemplificar lo anterior, considérese la consulta: "¿cuáles son los alumnos que pertenecen al mismo grupo en que se encuentra inscrito 'Tedd Codd'? y una posible solución

```
select    B.cuenta
from      inscrito A, inscrito B
where     A.cuenta = ( select cuenta
                       from   alumno
                       where  nombre = "Tedd Codd" )
          and
          A.grupo = B.grupo
```

SQL permite el uso de los conectores > any, < any, >= any, <= any, = any, como se puede apreciar en el siguiente ejemplo:

```
select    nombre
from      alumno
where     alumno.promedio > any
          (select alumno.promedio
           from   alumno, inscrito
           where  inscrito.grupo = 12 and
                  alumno.cuenta = inscrito.cuenta )
```

Las construcciones in, > any, > all, etc. nos permiten probar un valor simple contra los miembros de un conjunto completo. Ya que **select** genera un conjunto de tuplos, se puede comparar conjuntos para determinar si uno contiene a todos los miembros del otro. Tal comparación se realiza en SQL utilizando las construcciones **contains** y **not contains**. Considere la consulta "Determinar los clientes quiénes tienen una cuenta en todas las sucursales localizadas en Aguascalientes". Por cada cliente necesitamos ver si el conjunto de todas las sucursales

en que él tiene una cuenta contiene al conjunto de sucursales en Aguascalientes.

```
select   nombre_cliente
from     deposito A
where    ( select nombre_sucursal
           from   deposito B
           where  A.nombre_cliente = B.nombre_cliente)
         contains
         ( select nombre
           from   sucursal
           where  ciudad = "Aguascalientes")
```

SQL ofrece algún control sobre el orden en que los tuplos son desplegados. La cláusula **order by** produce que los tuplos del resultado aparezcan en el orden deseado. Para enlistar en orden alfabético a todos los clientes que tienen un préstamo en la sucursal de Aguascalientes, se puede escribir

```
select   nombre_cliente
from     préstamo
where    sucursal = "Aguascalientes"
order by nombre_cliente
```

SQL tienen además la habilidad de calcular funciones sobre grupos de tuplos utilizando la cláusula **group by**. El atributo dado en la cláusula **group by** es empleado para formar los grupos. Los grupos se formarán colocando en cada uno los tuplos con el mismo valor de atributo. Sobre los grupos se pueden calcular:

- promedio: avg
- mínimo: min
- máximo: max
- total: sum
- cuenta: count

Para encontrar el promedio de cuentas en todas las sucursales, se escribiría

```
select    nombre_sucursal, avg (balance)
from      deposito
group by  nombre_sucursal
```

Esta última consulta se puede complicar solicitando sólo aquellas sucursales que tengan un promedio mayor que 100 millones. La consulta emplea la cláusula **having**, que permite el uso de operadores agregados.

```
select    nombre_sucursal, avg (balance)
from      deposito
group by  nombre_sucursal
having    avg (balance) > 100
```

Finalmente, Si se desea contar el número de sucursales con promedio mayor que 100 millones, es posible emplear la notación **count (*)** para encontrar el número de tuplos en la relación:

```
select    count (*)
from      deposito
group by  nombre_sucursal
having    avg (balance) > 100
```

Las posibilidades de SQL son realmente mayores de los que hasta aquí se ha visto, sin embargo, requeriría un tema especial poderlas cubrir completamente.

página intencionalmente blanca